

Diplomarbeit

Mustergetriebene Anwendungsentwicklung

Gezeigt am Beispiel der Restrukturierung
des Eclipse-Plugins „Matrix4.plot“

zur Erlangung des akademischen Grades

Diplom Informatiker (FH)

vorgelegt dem Fachbereich Mathematik, Naturwissenschaften und Informatik
der Fachhochschule Gießen-Friedberg

Christian Gerhardt

28. Februar 2006

Referent: Prof. Klaus Quibeldey-Cirkel

Koreferent: Prof. Wolfgang Henrich

Inhaltsverzeichnis

Vorwort.....	3
1. Einleitung.....	5
1.1. Aufgabenstellung.....	6
1.2. Lesehinweise.....	7
2. Dotplots.....	8
2.1. Übersicht.....	8
2.2. Grundlegende Muster.....	9
2.3. Anwendungsgebiete.....	11
3. Eclipse.....	16
3.1 Übersicht.....	16
3.2 Plugins.....	16
3.3 Rich-Client-Plattform.....	17
4. Matrix4.plot.....	19
4.1 Übersicht.....	19
4.2 Ausgangssituation.....	19
5. Vorgehensweise.....	24
5.1 Extreme Programming.....	24
5.2 User Stories.....	24
5.3 Testfirstansatz.....	25
5.4 JUnit.....	26
6. Planung.....	27
6.1 Übersicht.....	27
6.2 Service-Framework.....	28
6.3 Pluginaufsatz für das Service Framework.....	29
6.4 Adaption des Plugin Framework für Matrix4.plot.....	31
6.5 Adaption der GUI auf des Plugin Framework.....	32
7. Umsetzung.....	33
7.1 Konzeptionelle Änderungen.....	33
7.2 Das Service Framework.....	37
7.3 Das Plugin-Framework.....	49
7.4 Das Kernsystem.....	57
7.5 Anpassungen an die Neue Architektur.....	65
7.6 Portierung auf die Rich Client Plattform.....	72
8. Resümee.....	76
8.1 Möglichkeiten der neuen Architektur.....	76
8.2 Erfahrungen mit Testfirst und JUnit.....	76
8.3 Ausblick.....	77
Anhang I.....	81
Anhang II.....	83

Vorwort

Die hier vorliegende Diplomarbeit ist der Abschluss meines Studiums und ein Wendepunkt in meinem Leben.

Das Ende meiner Studienzeit erfüllt mich mit Wehmut und ein wenig Trauer, aber nichtsdestotrotz auch mit der Genugtuung diesen manchmal schwierigen Weg erfolgreich hinter mich gebracht zu haben. Daher möchte ich an dieser Stelle all denjenigen Danken, die mich auf meiner Reise unterstützt haben.

Professor Quibeldey-Cirkel, für seine Geduld und sein Talent immer wieder neue Ideen einzubringen.

Michael Weiss, für seine immer nützlichen Tipps und Tricks und die Zeit die er in meine schreckliche Satzstruktur gesteckt hat.

Stephanie Keller, die mir gezeigt hat, was man an meinem Layout noch verbessern konnte.

Jutta Zinser und Alexander Rübin, für den letzten Schliff.

Und ich danke meinen Eltern, ohne deren jahrelange Unterstützung, ich nicht dort wäre, wo ich heute stehe.

Christian Gerhardt

26. Februar 2006

1. Einleitung

Der Inhalt dieser Diplomarbeit ist eine Untersuchung zur praktischen Anwendung gängiger Softwareentwicklungsmuster. Diese wurden beim Architekturreview und Redesign des Programms `Matrix4.plot`¹ eingesetzt und auf ihre Eignung hin untersucht.

Entwurfsmuster sind die Antwort der Informatik auf das immer wiederkehrende Dilemma, das Software für zwei verschiedene Problemstellungen zu zwei verschiedenen Softwarelösungen führen. Ein Textverarbeitungsprogramm lässt sich nur sehr schwer zur Prozesssteuerung einsetzen, und Versuche diese These zu widerlegen, führen zu der Einsicht, dass ein reines Prozesssteuerungsprogramm leichter und besser realisierbar ist.

Entwurfsmuster beschreiben sich wiederholende Konzepte in der Softwareentwicklung und werden meist mit Mitteln des objektorientierten Softwareentwurfs beschrieben. Ein Entwicklungsmuster bestimmt in diesem Fall die Beziehungen und das Verhalten zwischen den Objekten oder die Verwendung eines Objekts im System.

Es hat sich gezeigt, dass die Verwendung von Entwurfsmustern in der Anwendungsentwicklung, den Softwareentwurf in den Bereichen der Flexibilität und Qualität erheblich verbessern. Sie bauen auf den Erfahrungen vergangener Entwicklungsarbeit auf und berücksichtigen bereits gelöste Probleme oder geben Lösungswege vor.

Diese Diplomarbeit beschäftigt sich mit der Frage, wie man Software auf der Grundlage von Entwurfsmustern entwickelt.

Die Idee für diese Arbeit kam aus zwei Richtungen: Zum einen gehörte ich zu dem Entwicklungsteam, das die erste Version von `Matrix4.plot` als Plugin für die Eclipse-Plattform während des Schwerpunktpraktikums 2004 an der FH Gießen-Friedberg entwickelt hat, und zum anderen soll das damals geschaffene Plugin als Basis der Projektarbeit im neu geschaffenen Masterstudiengang der Hochschule dienen.

Bei Dotplots handelt es sich um eine Form der Visualisierung von Daten, die darauf abzielt, Muster innerhalb von Datenstrukturen sichtbar zu machen. Grundlage hierfür ist Position und Häufigkeit von auftretenden Informationseinheiten.

Entwickelt wurden Dotplots ursprünglich für die Genforschung, um sich wiederholende Gensequenzen vergleichen zu können. Aber das Konzept lässt sich auch auf andere Datentypen verwenden.

Im Kontext der Informatik sind besonders Quellcodeuntersuchungen in unterschiedlichen Programmiersprachen interessant. Aber auch aus normalen Texten kann `Matrix4.plot` Dotplots erstellen.

Hierzu wird ein Text (oder Quellcode) eingelesen und in eine Abfolge von Tokens

¹ sprich: Matrix-four-dot-plot

verwandelt. Diese Tokens können einzelne Wörter, ganze Sätze oder Kontrollstrukturen von Programmiersprachen sein. Nachdem die Tokens eine Reihe von Filtern durchlaufen, um die gewünschten Informationen weiter zu spezifizieren, werden sie in einer internen Datenstruktur, der F-Matrix, gespeichert. Sie enthält alle notwendigen Informationen wie Position, Häufigkeit und Gewichtung, um den Dotplot erstellen zu können.

Der tatsächliche Plot wird in einem Subsystem mit dem Namen Q-Image erzeugt, das die Tokens auf den Achsen einer zweidimensionalen Matrix aufspannt und bei einer Übereinstimmung der Tokens auf der X- und Y-Achse einen Pixel im Resultatsbild setzt. Dabei können die Texte, die auf die Achsen projiziert werden durchaus verschieden sein.

Das Resultatsbild ist der für diese Texte charakteristische Dotplot.

Das von uns erstellte Plugin leistet diese Grundfunktionalität mit dem Vorbehalt, dass es bei großen Datenmengen recht schnell an seine Leistungsgrenzen stößt. Rechenaufwand und Speicherverbrauch des Programms steigen durch den zweidimensionalen Charakter des Dotplots quadratisch zur Datenmenge der Eingabe.

Diesem Problem wurde in der weiterentwickelten zweiten Version des Programms dadurch angegangen, dass die Rechenlast durch Grid-Computing auf verschiedene Rechner verteilt wird. Die dabei erzielten Resultate sind sehr vielversprechend.

1.1. Aufgabenstellung

Ziel dieser Diplomarbeit ist es ein Architektur-Redesign des Matrix4.plot-Plugins durchzuführen. Dabei soll es auf die seit kurzem verfügbare Eclipse-Rich-Client-Plattform portiert werden um als eigenständiges Programm außerhalb der Eclipse-Entwicklungsumgebung nutzbar zu sein.

Das Architektur-Redesign soll dabei vor allem auf die zukünftige Aufgabe abzielen, als Basis für die Projektarbeit des Masterstudiengangs zu dienen.

Wichtig ist dabei:

- Klare Trennung der Programmstrukturen in Aufgabengebiete, um das Verständnis des Quellcodes und somit die Einarbeitung neuer Entwicklungsteams zu erleichtern.
- Die Programmstruktur soll flexibel durch einen Pluginmechanismus erweiterbar sein. Dadurch soll es möglich sein neue Programmmodule einzuführen und bestehende Module durch neue Funktionen zu erweitern.
- Die Erweiterung des Programms soll ohne Eingriff in bestehende Programmabläufe möglich sein, ohne dass dabei bestehender Code angetastet werden muss.
- Besondere Dienste des Programms, wie das Grid-Computing, sollen zentral für alle Programmmodule nutzbar sein.
- Die Eclipse Rich Client Plattform soll als Grundlage aller in Frage kommenden

Systemdienste genutzt werden. Das betrifft besonders die Verwaltung von Ressourcen und Konfigurationsinformationen, sowie die graphische Benutzeroberfläche.

1.2. Lesehinweise

Die im nachfolgenden vorgestellten Entwicklungs- und Architekturmuster wurden direkt aus den angegebenen Quellen entnommen und verstehen sich als direktes oder sinngemäßes Zitat.

Desweiteren wurden verschiedene Schriftkonventionen verwendet um Inhalte hervorzuheben.

- *Kursive Schrift* wurde benutzt um bestimmte Begriffe hervorzuheben. Das trifft besonders auf Namen von Plugins und Mustern zu.
- `Nichtproportional Schrift` kennzeichnet Quellcode als Ganzes und Klassen- und Methodennamen innerhalb von Text. Auch XML Elemente und Attribute werden auf diese Weise hervorgehoben.

2. Dotplots

2.1. Übersicht

Dotplots finden ihren Ursprung in der Genetik, in der sie zum Vergleich von Gensequenzen entwickelt wurden. Dotplots erleichtern dabei die Suche nach Gemeinsamkeiten in der Erbinformation von verschiedenen Lebewesen mit dem Ziel, Verwandtschaftsbeziehungen und Erbkrankheiten besser zu verstehen. Auch lassen sich Rückschlüsse auf die Funktionsweise einzelner Gene ziehen, wenn sie bei verschiedenen Spezies die gleiche Aufgabe erfüllen. Ein Dotplot vergleicht hierzu die Anordnungen der genetischen Grundbausteine, die Aminosäuren Guanin, Cytosin, Adenin und Thymin, miteinander. [DTP]

Verallgemeinert man das Prinzip kann man die einzelnen Aminosäuren auch als Wörter einer Sprache ansehen. In diesem Kontext sind Wörter die kleinsten bedeutungsbehafteten Bausteine einer Sprache, die im folgenden Tokens genannt werden. Ein Dotplot vergleicht demnach Abfolgen von Tokens miteinander.

Um einen Dotplot zu erzeugen wird eine zweidimensionale Matrix aufgespannt an deren X und Y-Achse die Abfolge der zu vergleichenden Tokens aufgereiht sind. Sind beide Abfolgen identisch, wird von einem Selbstvergleich gesprochen. Als nächstes wird an jeder Stelle der Matrix (i, j) , an der die Tokens T_i, T_j gleich sind, eine Markierung gesetzt. Als letzter Schritt wird die Matrix in ein Bild umgewandelt, in dem jede Markierung durch einen Bildpunkt repräsentiert wird. Dieses Bild wird Dotplot genannt.

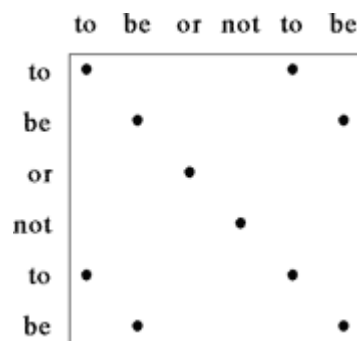


Abbildung 1 - Dotplotbeispiel
[DSP]

Bei näherer Betrachtung solcher Dotplots fällt auf, dass sich nur schwer interessante Bereiche herausuchen lassen. Das entsteht dadurch, dass häufig vorkommende Tokens eine Menge an Treffern erzeugen, die interessante Bereiche des Dotplots verdecken. In der deutschen Sprache wären das zum Beispiel Personalpronomen.

Um diesem Problem zu begegnen kann man auf den Umwandlungsalgorithmus, mit dem der Dotplot aus der Matrix erzeugt wird, Einfluss nehmen. In dem für jedes Token eine Gewichtung angegeben wird, können wichtige Tokens hervorgehoben und unwichtige vernachlässigt werden. Dies kann automatisch

geschehen, indem häufige Tokens eine geringere Gewichtung erhalten, oder per Hand durchgeführt werden, indem die Gewichtung von ausgewählten Tokens erhöht wird.

2.2. Grundlegende Muster

Nach der Aufarbeitung des Dotplots auf diese Weise lassen sich schnell typische Muster erkennen. Charakteristisch ist die Hauptdiagonale, die im Selbstvergleich immer vorhanden ist.

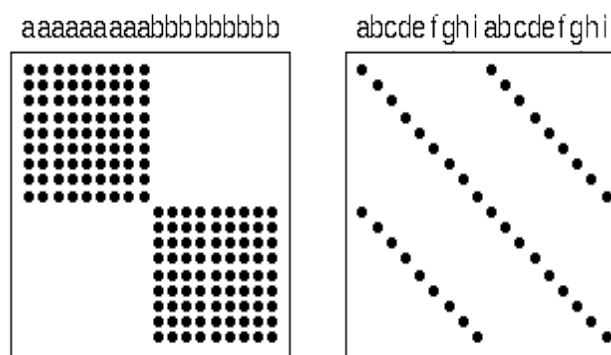


Abbildung 2 - Grundlegende Muster[DSP]

Die beiden grundlegenden Muster sind Quadrate und Diagonalen. Quadrate entstehen durch lokale Wiederholungen gleicher Tokens, und Diagonalen durch die Wiederholung von gleicher Abschnitte.[DP]

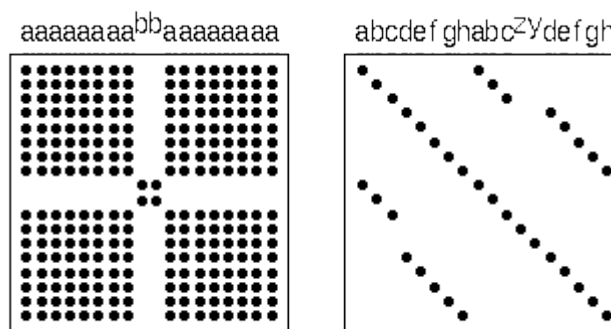


Abbildung 3 - Störungen in Mustern[DSP]

Das Auftreten dieser Muster in reiner Form geschieht nur selten und ist unter normalen Umständen auch nicht so interessant. Störungen und Variationen in diesen Mustern geben wesentlich mehr Auskunft über den zu untersuchenden Datensatz. Störungen in Mustern deuten auf das Einfügen neuer Tokens in den Datensatz hin. [DP]

Vergleicht man einen Text mit einer früheren Version des Textes, so deuten abgeknickte oder unterbrochene Diagonalen auf Stellen hin, an denen der Text verändert wurde. Eine abgeknickte Diagonale entsteht durch das Entfernen von Tokens und eine unterbrochene entsteht durch das Einfügen von Tokens.

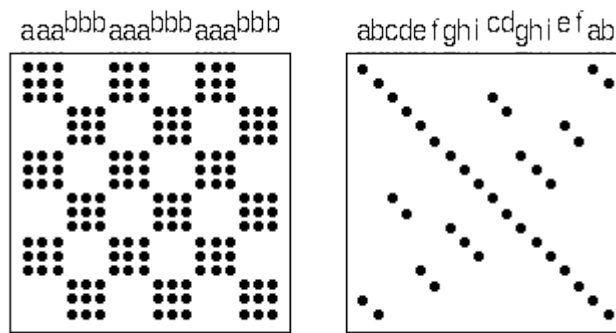


Abbildung 4 - Reorganisation von Blöcken und Diagonalen [DSP]

Die beiden hier gezeigten Muster entstehen durch Reorganisation der Datensätze, wenn zum Beispiel Absätze oder Kapitel eines Textes neu angeordnet werden.[DP]

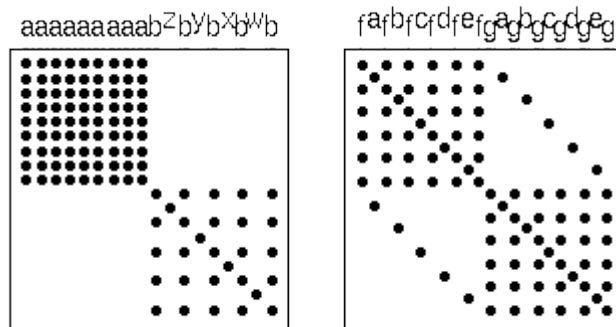


Abbildung 5 - Kombination von Mustern [DSP]

Genauso wie Muster selten in reiner Form angetroffen werden, treten sie separat, sondern meist in Kombination auf. Typisch sind die beiden oben gezeigten Beispiele der Kombination von Blöcken und Diagonalen.[DP]

2.3. Anwendungsgebiete

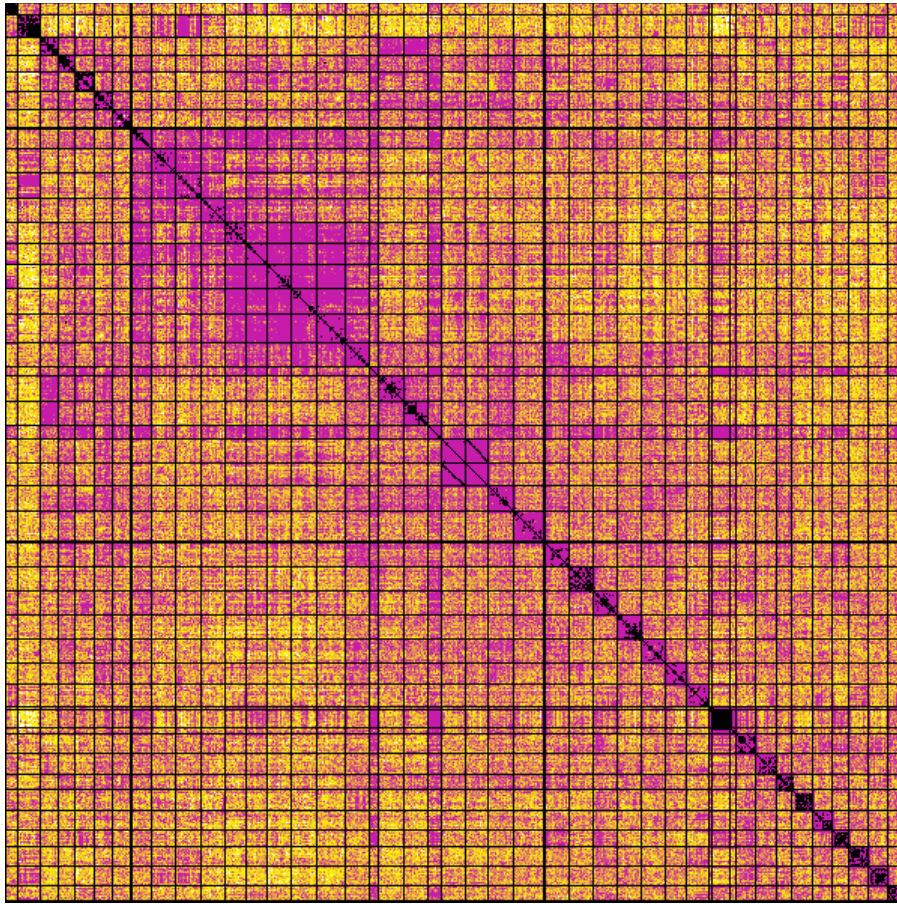


Abbildung 6 - Dotplot aller Werke von Shakespeare [DSP]

Neben seiner Bedeutung in den bereits angesprochenen Bereich der Genetik, finden Dotplots ihre Hauptanwendung in den Bereichen der Analyse und Optimierung von Informationen.

Angewandt auf Sprachen lassen sich verschiedene Übersetzungen von Texten untereinander vergleichen. So gibt es gewisse Grundähnlichkeiten von Sprachen innerhalb einer Sprachfamilie, wie zum Beispiel die romanischen oder slawischen Sprachen, die sich mit Dotplots gut verdeutlichen lassen.[Pla]

Mit Hilfe dieser Information kann die Qualität einer Übersetzung überprüft werden. Zwei Übersetzungen eines Textes sollten eine gleiche Ähnlichkeit untereinander besitzen, wie für die beiden Sprachen typisch ist.[DTP]

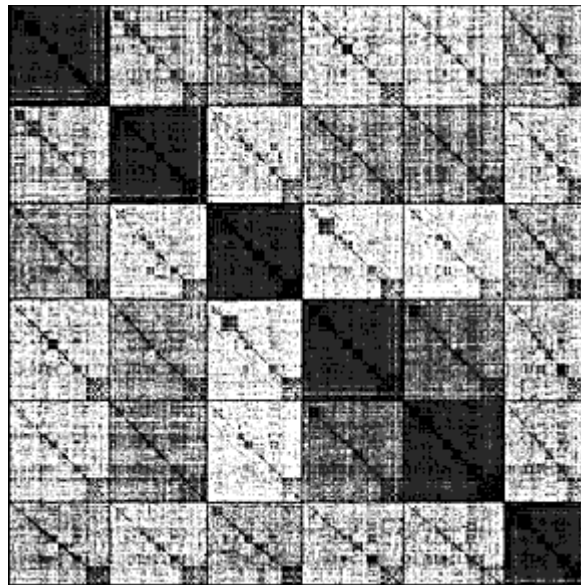


Abbildung 7 - Vergleich von Übersetzungen [DSP]

Dieses Beispiel ist ein Dotplot bestehend aus einem Vergleich der sechs Übersetzungen eines technischen Handbuchs miteinander: von links nach rechts Dänisch, Französisch, Deutsch, Italienisch, Spanisch und Schwedisch.

Um eine bessere Aussage des Dotplots zu erhalten wurden zu diesem Zweck jeweils vier Wörter zu Gruppen in einem einzelnen Token zusammengefasst. Diese Gruppen oder 4-Gramme, sind eine sprachwissenschaftliche Methode, um eine größere Ähnlichkeiten zu erhalten.

Im Dotplot stellt jeder zu erkennende Block einen einzelnen Vergleich zweier Übersetzungen dar. Die dunklen Quadrate der Hauptdiagonale sind die Selbstvergleiche, die naturgemäß eine besonders hohe Ähnlichkeit besitzen. Je dunkler die Färbung eines Blocks, desto mehr die Ähnlichkeit der Übersetzungen, und desto größer sind die Sprachen miteinander verwandt. [Pla]

Die ebenfalls erkennbaren Diagonalen innerhalb der Blöcke entstehen durch in allen Sprachen einheitliche Begriffen, Namen, Zahlen und nicht übersetzte Wörter.

Als nächstes zu erwähnen ist die Plagiaterkennung, die mit Dotplots ein visuelles Werkzeug in die Hand bekommt. Auch hier werden 4-Gramme verwendet, um die Ähnlichkeit eines vermeintlichen Plagiats mit dem Original zu ermitteln und die visuelle Auswertung verbessert die Effektivität der Analyse.[Pla]

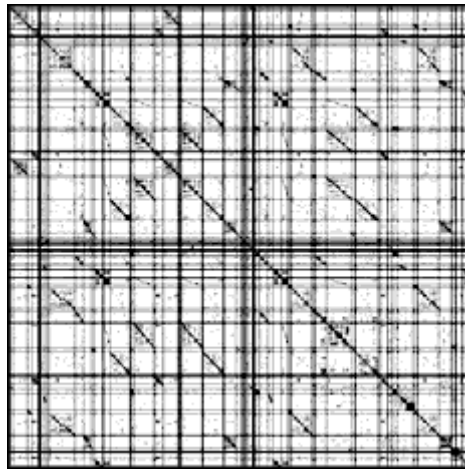


Abbildung 8 - Versionsvergleich [DSP]

Das obige Beispiel entstand durch den Vergleich zweier Versionen eines Textes. Klar zu erkennen sind die abgeknickten und unterbrochenen Diagonalen, die durch die Änderungen an der neueren Version entstanden sind.

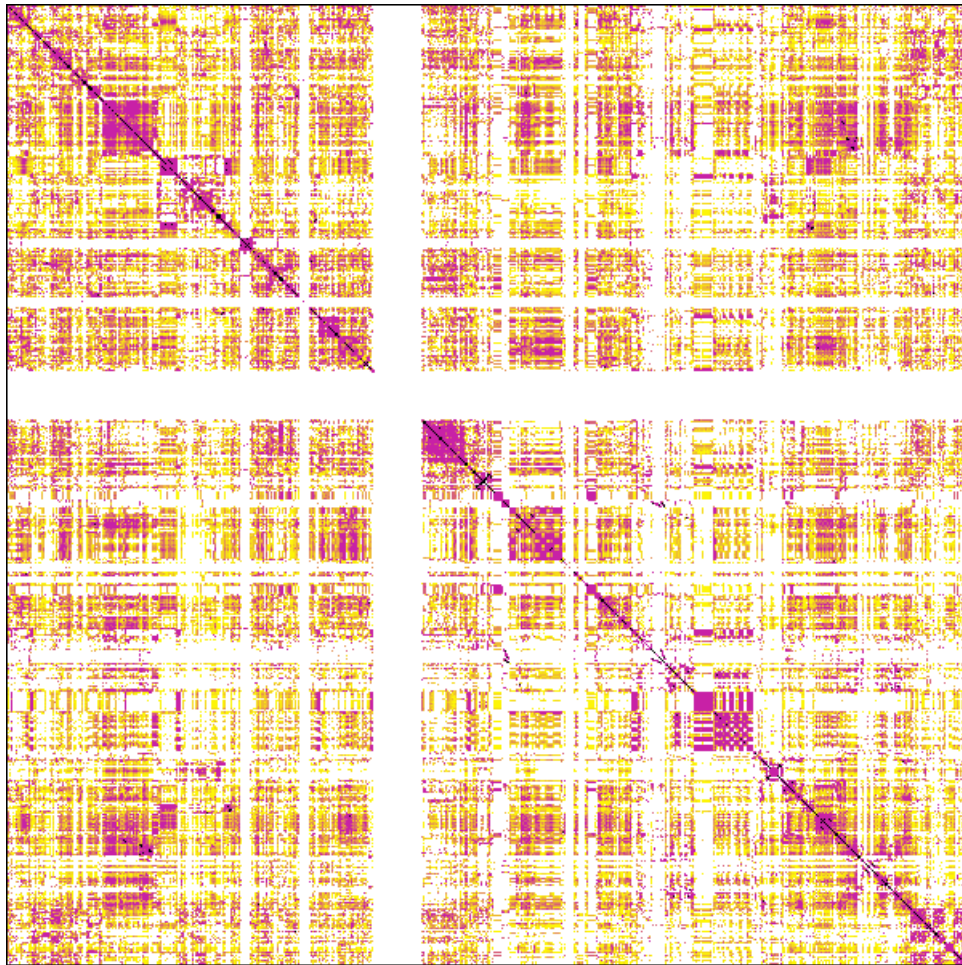


Abbildung 9 - Vergleich von 290.000 Dateinamen[DSP]

Dieser Dotplot entstand durch den Vergleich von 290.000 Dateinamen eines Servers. Die dunklen Bereiche kennzeichnen redundante Dateinamen, die durch verschiedene Anwendungen verursacht werden, die von Benutzern des System parallel und in unterschiedlichen Versionen installiert wurden. Das große helle Kreuz entstand durch temporäre Dateinamen, die automatisch angelegt wurden als eine Anwendung zur Übermittlung an ein anderes System in kleinere Dateien aufgespalten wurde.

In der Informatik spielen Dotplots gleich an mehreren Stellen eine Rolle.

Während des Refactoring von Quellcode kann ein Dotplot helfen doppelten Code aufzuspüren. Durch Refactoring wird Quellcode optimiert indem mehrfach vorkommender Code zusammengefasst wird und komplexe Strukturen durch einfachere ersetzt werden. Dadurch wird die Wartung des Quellcodes und das Auffinden von Fehlern vereinfacht. Die Funktionsweise des Codes bleibt beim Refactoring unangetastet.

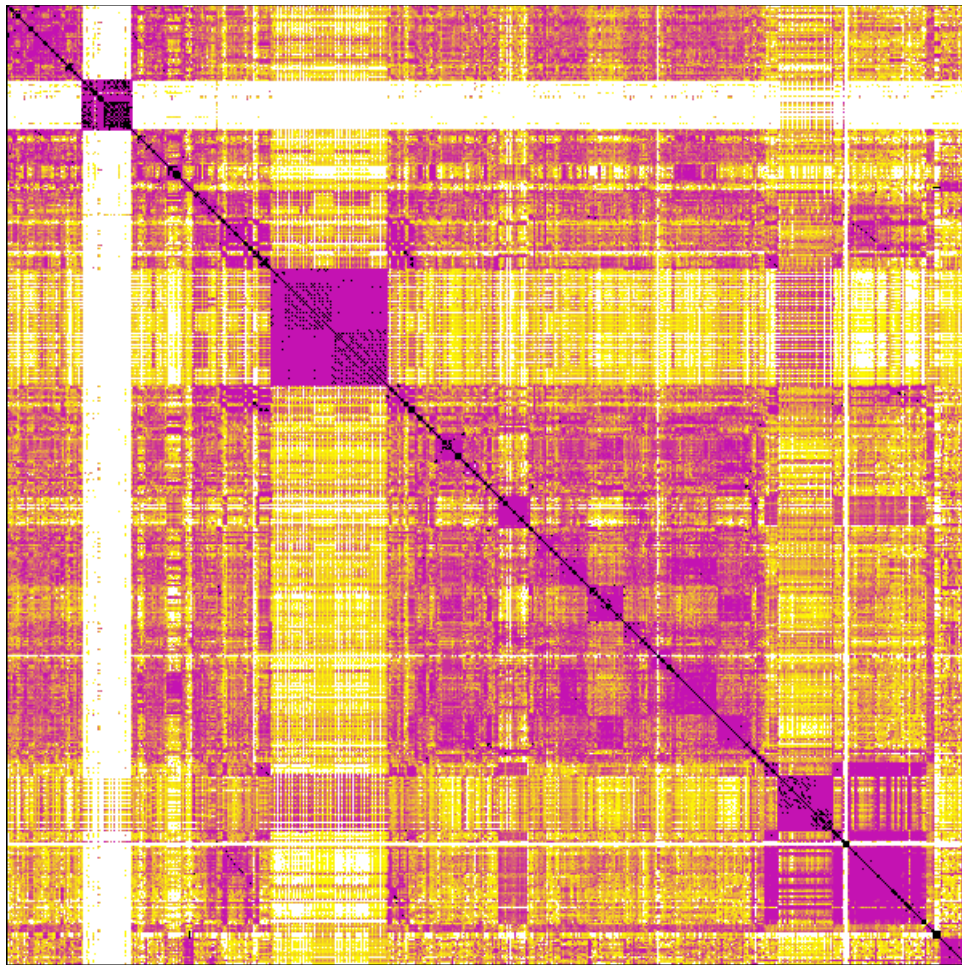


Abbildung 10 - 2 Millionen Zeilen C-Code [DSP]

Der oben gezeigte Dotplot entstand durch den Quellcode der Software eines Telekommunikations Switches aus zwei Millionen Zeilen C-Code. Die dunkleren Bereiche zeigen eine hohe Trefferdichte an. Dunkle Bereiche entlang der Hauptdiagonalen entstanden durch Submodule und dunkle Bereiche abseits von ihr zeigen hohe Ähnlichkeit zwischen einzelnen Submodulen. Die leicht zu erkennenden großen dunklen Quadrate werden durch Redundanzen in der Initialisierung von Signaltabellen und Zustandsautomaten erzeugt.

Auch das Beheben von Programmfehlern wird erleichtert, da Stellen mit dem gleichen fehlerhaften Code durch einen Dotplot leichter gefunden werden können.

Genauso können Dotplots beim Design neuer Programmiersprachen hilfreich sein. Ein Dotplot kann helfen häufig wiederkehrende syntaktische Elemente der neuen Programmiersprache zu finden, um sie dann auf ihre Optimierbarkeit hin zu untersuchen.[DTP]

3. Eclipse

3.1 Übersicht

Eclipse basiert ursprünglich auf einer erweiterbaren Entwicklungsumgebung für die Programmiersprache Java und entwickelte sich mit der Zeit zu einer vielseitigen Plattform für alle möglichen Arten von Programmen. Eclipse selbst ist in Java implementiert und profitiert dadurch von der Plattformunabhängigkeit dieser Programmiersprache. Zudem verfügt Eclipse über das ebenfalls plattformunabhängige Standard Widget Toolkit (SWT), das eigens für die Gestaltung der graphischen Oberfläche von Eclipse entwickelt wurde. [EC], [RCE]

Von Haus aus wird Eclipse zusätzlich zur obligatorischen Entwicklungsumgebung für Java, auch mit einer Reihe nützlicher Tools ausgestattet, wie JUnit², Apache ANT³ und einem CVS⁴-Client.

Die Benutzeroberfläche von Eclipse, die Workbench, ist in drei Basiselemente aufgeteilt. Diese sind Editoren, Views und Perspektiven.

Ein Editor ist das grundlegende Arbeitselement mit dem Dateien bearbeitet werden können. Sie werden direkt von anderen Bedienelementen wie Menüs und Toolbars beeinflusst.

Ein View ist unabhängiger und kann für jede erdenkliche Art der Benutzereingabe und der Anzeige von Informationen genutzt werden.

Perspektiven schaffen Ordnung und gruppieren Views und Editoren thematisch. So gibt es Perspektiven zur Javaentwicklung, Ressourcenmanagement und CVS-Synchronisation.

Ein weiteres Grundkonzept von Eclipse ist der Arbeitsbereich. Hierbei handelt es sich um ein spezielles Verzeichnis, in dem Eclipse eigenständig die Dateien des Benutzers in Form von Projekten verwaltet.

3.2 Plugins

Alles in Eclipse ist ein Plugin. Das bedeutet, dass jede Funktionalität, die dem Anwender zur Verfügung gestellt wird, in Form eines Plugins zum Kern von Eclipse hinzugefügt worden ist. Dieser Kern ist ein System zur Verwaltung von Plugins, der im Vergleich zum Lieferumfang relativ winzig ist. Durch dieses zentrale Konzept bekommt Eclipse seine Flexibilität und freie Erweiterbarkeit. So ist natürlich die Javaentwicklungsumgebung ein Plugin, genauso wie JUnit, aber auch das Windowingframework SWT ist als Plugin in Eclipse eingebunden. Hieran kann man erkennen, dass ein Plugin nicht zwangsläufig eine Perspektive oder einen Editor besitzen muss. Das Plugin kann auch einfach nur Dienste für andere Plugins bereitstellen, die sich der Wahrnehmung des Benutzers entziehen.[RCE]

Dadurch entstehen Abhängigkeitsketten von Plugins, um deren Auflösung sich auch

2 Siehe auch: [JU]

3 Siehe auch: [Ant]

4 Siehe auch: [CVS]

Eclipse kümmert. So benötigt zum Beispiel das JFace-Plugin, dass verschiedene Oberflächenelemente wie Wizards und Dialoge vereinheitlicht, das SWT-Plugin, um diese zu erzeugen. Genauso brauchen andere Plugins das JFace-Plugin zur Erzeugung ihrer konkreten Wizards in der Anwendung.

Die Verwaltung eines Plugins wird über eine XML-Datei durchgeführt, in der alle Einstellungen des Plugins festgehalten werden. Der Name dieser Datei ist `plugin.xml` und ist für alle Plugins gleich, damit Eclipse die Datei identifizieren kann. Zu den Einstellungen in dieser Datei gehören die Jar-Dateien mit dem eigentlichen Programmcode, zusätzliche Ressourcen, wie z.B. Bilder, Icons oder zusätzliche Programmbibliotheken. Auch Hilfeseiten auf Basis von HTML und Dokumentationen werden in Eclipse über diese Datei eingebunden.

Ein weiteres Feature sind die Plugin-Extensionpoints, die ebenfalls über die XML-Datei eingestellt werden. Sie bilden den Kern der von Eclipse betriebenen Erweiterungspolitik. Dadurch ist es möglich nicht nur die Eclipse Plattform über Plugins zu erweitern, sondern auch die Plugins selber über Extensionpoints erweiterbar zu machen. Tatsächlich handelt es sich bei den meisten Plugins um Erweiterungen von bestehenden Plugins.

Den Extensionpoint `org.eclipse.ui.perspectives` benutzt `Matrix4.plot` so zum Beispiel, um seine Perspektive anzumelden, `org.eclipse.ui.views` um seine Views zu integrieren und `org.eclipse.ui.actionSets`, um neue Einträge in der Menü- und Werkzeugleiste zu machen.

Wenn man Eclipse um neue Funktionen erweitern möchte, wie zum Beispiel Eclipse für eine neue Programmiersprache als Entwicklungsumgebung zu nutzen, reicht ein einzelnes Plugin in der Regel nicht aus, wenn neue Editoren, Debugger, Compiler usw. eingefügt werden müssen. Hierzu bietet Eclipse die Möglichkeit mehrere Plugins zu einem Feature zusammenzufassen, mit dem die Plugins als Verbund ausgeliefert werden.

3.3 Rich-Client-Platform

Die Rich-Client-Platform wurde mit Version 3.1 in Eclipse eingeführt. Ausgehend von der Prämisse, dass der Kern von Eclipse recht klein ist, und der Großteil der mitgelieferten Plugins nur für eine bestimmte Zielgruppe von Anwendern interessant ist, kam die Idee auf, Eclipse nicht als Plattform für eine Menge von Programmen zu benutzen, sondern nur für ein einzelnes. Dadurch bekam Eclipse die Möglichkeit für das neue Anwendungsgebiet der Rich Clients interessant zu werden [RCE].

Mit der in der heutigen Zeit anzutreffenden Vernetzung von verschiedensten datenverarbeitenden Plattformen, wie PCs, Handys, PDAs, usw. ist es nötig, auch eine Benutzeroberfläche für Anwendungen zu schaffen, die auf all diesen Plattformen laufen soll. Der übliche Weg dazu ist eine HTML-Lösung zu suchen, um die auf allen Plattformen verfügbare Browsertechnologie, für die eigene Anwendung zu nutzen. Jeder der einmal mit HTML gearbeitet hat weiß, dass solche grafischen Benutzeroberflächen alles andere als Benutzerfreundlich sind. Die Versuche die Einschränkungen von HTML durch zusätzliche Applikationen

wie Java Applets, Macromedia Flash oder Active-X zu erweitern, machen es wieder notwendig, dass diese Applikationen auch auf den Zielplattformen verfügbar sind.[RCE]

Die Lösung dieses Dilemmas sind Rich Clients, benutzerfreundliche graphische Oberflächen, die auf möglichst vielen Plattformen verfügbar sind. Die Eclipse-Rich-Client-Plattform nutzt den Umstand, dass viele Plattformen über eine Java Virtual Machine verfügen und ein Rich Client, auf Basis von Eclipse, auf ihnen ohne Probleme lauffähig ist.

Die Rich-Client-Plattform reduziert Eclipse auf seine fundamentalen Plugins, wie SWT und JFace, oder das Help-Plugin. Diese Plugins sind die Basis für den neu entstehenden Rich Client. Der Umfang von Eclipse reduziert sich dadurch deutlich um etwa 90%.

4. Matrix4.plot

4.1 Übersicht

Das Eclipse-Plugin Matrix4.plot entstand während des Schwerpunktpraktikums im Sommersemester 2004. Es wurde von Professor Quibeldey-Cirkel mit dem Ziel initiiert ein Dotplot-Programm auf Basis der Eclipse Plattform zu erstellen. Für die Mitglieder des Teams war sowohl die Materie der Dotplots als auch das Medium der Eclipse Plattform unbekannt. Somit lag das Ziel eher darin die Aufgabe im gesetzten Zeitraum umzusetzen, als sich über nachfolgende Projekte Gedanken zumachen. Dies geschah erst in der Bewertung der Arbeit durch Professor Quibeldey-Cirkel, der darin weiterführendes Potenzial für zukünftige Diplom- und Masterarbeiten sah.

Die erste Diplomarbeit zu diesem Thema wurde von Tobias Gesellchen[VgD] im Sommersemester 2005 fertig gestellt und erste Praktika des Masterstudiengangs begannen im Wintersemester 2005/2006.

Seit Juni 2005 wird Matrix4.plot zudem als Open Source Projekt bei Sourceforge [SF] einer breiten Öffentlichkeit angeboten.

In diesen Hintergrund wird sich auch diese Diplomarbeit einreihen und die Arbeit fortführen.

4.2 Ausgangssituation

Matrix4.plot befand sich nach dem Schwerpunktpraktikum noch in einem rudimentären Zustand. Seine Möglichkeiten beschränkten sich auf die Grundfunktionen der Erstellung von Dotplots aus Textdateien.[M4DP]

Hierzu wird die Datei zunächst eingelesen und von einem Scanner in einen Tokenstrom umgewandelt. Bei einem Scanner handelt es sich um ein Teilprogramm, dass sich um das Einlesen der Datei und deren Interpretation kümmert. Dabei werden die eingelesenen Buchstaben anhand der im Scanner angegebenen Regeln in Tokens zusammengefasst. Ein Token ist die kleinste Sinn gebende Einheit eines Textes. Dies können Buchstaben, Worte, Phrasen oder ganze Sätze sein. Die in Matrix4.plot benutzten Scanner wurden alle automatisch mit dem Tool *JFlex* [JFlex] erstellt, das vor allem in der Erstellung von Compilern für Programmiersprachen Verwendung findet.

Der erzeugte Tokenstrom liefert ein gelesenes Token nach dem anderen, die danach in die F-Matrix des Dotplots weiterverarbeitet werden.

Da die ersten Anwender Informatikern waren, lag das Hauptinteresse in Dotplots von Quellcode, weshalb Scanner für die gebräuchlichsten Programmiersprachen C/C++, *Java* und *PHP* eingebaut wurden.

Die Besonderheit des Tokenstroms ist es, dass sich ohne Probleme mehrere Tokenströme hintereinander verketteten lassen, ohne dass dies die Weiterverarbeitung beeinträchtigt. Ein Tokenstrom kann seine Tokens auch von einem anderen Tokenstrom erhalten, was es ermöglicht den vom Scanner gelieferten

Tokenstrom zu filtern oder umzuwandeln.

Die in Matrix4.plot implementierten Filter ermöglichen es, einzelne Token oder ganze Tokenarten aus dem Tokenstrom zu entfernen, oder alle Token einer Zeile bzw. eines Satzes zu einem einzelnen Token zusammenzufassen. Damit lassen sich Textzeilen oder Sätze miteinander vergleichen.

Durch die Verkettung von Tokenströmen lassen sich auch mehrere Filter hintereinander schalten.

Die Token eines Tokenstroms werden nun in eine F-Matrix weiterverarbeitet, die die Grundlage des Dotplots liefert. Der Begriff F-Matrix hat sich an dieser Stelle gehalten, da er sich leicht mit der Theorie assoziieren lässt. Die Theorie besagt, dass es sich bei der F-Matrix um eine Matrix aus Fließkommawerten handelt, in der Position und Gewichtung von Tokens festgehalten werden. In der Praxis hat sich dieses Prinzip nicht bewährt, da in einer Matrix von fester Größe auch Platz für nicht vorhandene Treffer (weiße Stellen im Dotplot) reserviert werden muss.

Innerhalb von Matrix4.plot wird der Begriff F-Matrix daher für den Programmteil verwendet, der die Tokenvergleiche durchführt und die Treffer festhält. Eine Datenstruktur im Sinne einer Matrix liegt nicht mehr vor.

Nach der Analyse des Tokenstroms und der Erzeugung der F-Matrix-Datenstrukturen wird der eigentliche Dotplot erstellt. Der Programmteil, der dies bewerkstelligt wird Q-Image genannt. Das Q-Image baut den Dotplot mit Hilfe der in der F-Matrix gespeicherten Übereinstimmungen von Tokenpositionen auf. Dabei greift es auf eine mitgeführte Tokenstatistik zurück, um die Einfärbung anhand von Häufigkeiten und gewählten Farbschemas durchzuführen. Dadurch können selten vorkommende Token hervorgehoben werden, die sonst zwischen vielen Treffern untergehen würden.

Matrix4.plot bietet mehrere gängige Bildformate als Exportmöglichkeit an, die mit Hilfe der Java Advanced Imaging (JAI) Programmbibliothek umgesetzt werden.

Durch das quadratische Verhalten der Dotplots (1000 Token erzeugen ein 1000 x 1000 Bildpunkte großen Dotplot) stößt das Programm schnell an die Grenze des Hauptspeichers. Dies geschieht besonders schnell beim Vergleich mehrerer Dateien. Dieser Umstand war es, der die erste zum Thema Dotplot geschriebene Diplomarbeit initiierte. Tobias Gesellchen erweiterte Matrix4.plot um den Information Mural Algorithmus und die Möglichkeit des Grid-Computing. Beide Verfahren zielen darauf ab dem Speicherhunger des Programms gerecht zu werden.

Der Information Mural Algorithmus ist ein Kompressionsverfahren, bei dem Bildinformationen nicht verloren gehen. Normale Kompressionsverfahren fassen mehrere Bildpunkte zusammen und ermitteln den Durchschnitt ihrer Farbwerte als Farbwert für den neuen Bildpunkt. Nimmt man als Beispiel ein monochromes Bild

Farbwert: 0 = Weiß, Farbwert: 1 = schwarz

würde beim Zusammenfassen von vier Bildpunkten bei dem nur ein Bildpunkt gesetzt ist, dieser bei einer normalen Kompression verschwinden

Farbwerte: $(1 + 0 + 0 + 0) / 4 = 1 / 4$ abgerundet = 0 !

und die Information verloren gehen. Der Information Murrel erhält diese Information und würde hier dem neuen Pixel den Farbwert 1 zuweisen. [VgD]

Die Speicherplatzersparnis ist im Vergleich zu vorher gewaltig, aber noch nicht ausreichend. Daher wurde zusätzlich das Grid-Computing eingeführt, bei dem die Berechnung des Dotplots auf mehrere Rechner verteilt wird und die Verarbeitungskapazität der zu bewältigenden Aufgabe angepasst werden kann.

Die Eclipse-Plattform wird zu diesem Zeitpunkt nur als Framework für die Benutzerschnittstelle genutzt. Man kann Dateien auswählen und sich deren Dotplot anzeigen lassen. Bereits die Anzeige der Konfiguration ist unabhängig von Eclipse realisiert und benutzt nur das Eclipse beiliegende Standard Widgets Toolkit (SWT).

Muster: *Pipes & Filters*

Quelle: [PoSa]

Zweck

Das Pipes & Filters Muster bietet eine Struktur für Systeme, die Datenströme verarbeiten. Jeder Verarbeitungsschritt ist dabei in einem Filter gekapselt. Daten werden durch Kanäle⁵ von Filter zu Filter weitergegeben. Die Filter können immer neu angeordnet werden. Dies ermöglicht Familien verwandter Systeme zu bilden.

Struktur



Diagramm 1 - Struktur von Pipes & Filters

Teilnehmer

- **Filter**
 - holt Eingabedaten
 - führt Funktionen auf Eingabedaten aus
 - stellt Ausgabedaten bereit
- **Pipe**
 - überträgt Daten
 - speichert Daten zwischen
 - synchronisiert aktive Nachbarn
- **Data Source**
 - liefert Input an die verarbeitende Pipeline
- **Data Sink**
 - verbraucht Ausgaben

⁵ engl. Pipes

Pipes & Filters in der Anwendung

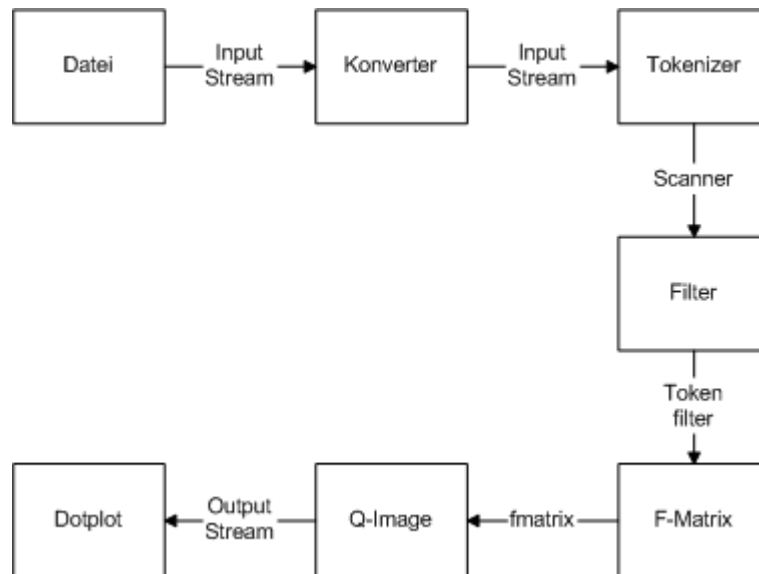


Diagramm 2 - Pipes & Filters in der Anwendung

Hier kann man schematisch verfolgen wie Matrix4.plot einen Dotplot erstellt. Die einzelnen Subsysteme verarbeiten und transformieren die Daten der Datei nach und nach zum fertigen Dotplot.

Die Verarbeitung der Daten erfolgt dabei auf unterschiedliche Weise.

- Der Konverter liest zunächst die gesamte Datei ein und schreibt die konvertierten Daten in eine neue Datei. An den Tokenizer übergibt er einen neuen `InputStream`.
- Der Tokenizer liest an dieser Stelle keine Daten ein, sondern erzeugt einen `Scanner`, der die Daten auf Verlangen ausliest (*pull*-Verfahren [PoSa])
- Auch der Filter liest noch keine Daten, sondern erzeugt einen `Filter` der ebenfalls auf Verlangen den `Scanner` benutzt um ein Token zurückzugeben. Dieses Token wird den Filterregeln unterworfen.
- Die F-Matrix liest die Daten in dem sich die Tokens vom Filter anfordert komplett ein, und gibt eine `fmatrix`-Datenstruktur zurück.
- Das Q-Image benutzt die `fmatrix`-Datenstruktur um den Dotplot zu erzeugen.

5. Vorgehensweise

5.1 Extreme Programming

Extreme Programming ist ein leichtgewichtiger iterativer Entwicklungsprozess der sich besonders für kleinere Projekte eignet. Sein Ziel ist es die große Menge an anfallenden Dokumenten anderer, schwergewichtigere Entwicklungsprozesse zu vermeiden und das Risiko für Kunden und Entwickler zu minimieren. Hierzu wird der Kunde von Anfang an stark in den Entwicklungsprozess integriert mit dem Ziel die Kundenwünsche in den Vordergrund zu stellen. [XP]

Es ist Teil der Philosophie von Extreme Programming immer und zu jeder Zeit auf die Wünsche des Kunden einzugehen, selbst dann, wenn diese kurz vor Projektabschluss gemacht werden. Da es sich um einen iterativen Prozess handelt, in dem sich Planungs- und Umsetzungsphasen in kurzer Zeit wiederholen, kann schnell auf Kundenwünsche eingegangen werden.

Durch sein Leichtgewicht, bot sich Extreme Programming zur Umsetzung dieser Diplomarbeit an. Da Extreme Programming aber auf Entwicklungsteams zugeschnitten ist, sind nur einige der in Extreme Programming beschriebenen Methoden sinnvoll umsetzbar.

Umgesetzt wurden der Zentral gesetzte Testfirstansatz und die User Storries.

5.2 User Stories

Eine User Story ist in dem Bereich der Anwendungsfälle (Use Case) angesiedelt, in denen es um die Beschreibung von Systemverhalten auf Benutzereingaben geht. Dabei ist die User Story einfach betrachtet eine Verallgemeinerung des klassischen Use Case.[XP]

Klassische Use Cases beschreiben detailliert und vollständig wie das System auf Benutzereingaben reagiert. Insbesondere positive und negative Ereignispfade mit klar definierten Anfangs- und Endzuständen. So ist festgelegt, wann das System zur Ausführung eines Use Case kommt, was die Vorbedingungen sind, welche Eingaben der Benutzer tätigen muss, um in den gewünschten Endzustand zu kommen und was das System in einem eventuellen Fehlerfall zu tun hat.

Die beiden Schlagwörter auf die es ankommt sind *detailliert* und *vollständig*. Das gesamte Systemverhalten muss in allen Einzelheiten durch einen Use Case beschrieben sein!

So kann es durchaus vorkommen, dass ein komplexes System durch mehrere hundert Use Cases beschrieben wird.

User Stories verfolgen einen allgemeineren Ansatz, in dem sie das Ziel, mit dem das System benutzt wird, beschreiben. Im Vergleich zum klassischen Use Case, hören User Stories genau dort auf, wo der Detailreichtum beginnt und der Kunde eventuell die Übersicht verliert.

User Stories sind für die Zusammenarbeit mit dem Kunden sehr wichtig, da sie das Instrument sind, mit dem er direkten Einfluss auf die entstehende Software nehmen

kann. Daher sind sie in der Sprache des Kunden gehalten, ohne auf technische Fragen der Umsetzung einzugehen.

Erstellt werden User Stories in direkter Zusammenarbeit zwischen Entwickler und Kunde, in der die Systemparameter abgesteckt werden. Die Aufgabe des Kunden ist es das gewünschte Systemverhalten zu beschreiben und eine Gewichtung für das Gesamtprojekt anzugeben, mit der die User Stories für die anschließende Entwicklungsarbeit priorisiert werden.

Der Entwickler schätzt den Umfang und das Risiko der Umsetzung der User Story ab, um bereits hier einen Einblick für die auf den Kunden zukommenden Kosten zu erhalten. Auch kann noch Einfluss auf diese beiden Parameter genommen werden, in dem die User Story bei zu großem Umfang aufgeteilt wird, oder auf zu riskante Funktionen verzichtet wird.

Nachdem die User Story in Zusammenarbeit mit dem Kunden fertig gestellt wurde, beginnen die Entwickler mit der Festlegung von Tasks. Eine Task ist eine konkrete Programmieraufgabe, die von einem Programmiererpaar in etwa zwei, besser nur einem, Arbeitstag umgesetzt werden kann.

Diese Tasks sind für den Kunden eher uninteressant und dienen innerhalb des Entwicklungsteams der Planung der konkreten Projektarbeit. Dadurch, dass immer zwei Programmierer (Pair Programming) einer Task zugewiesen sind, wird verhindert, dass beim Ausfall eines Programmierers das Wissen um die Besonderheiten seiner Arbeit für das Team verloren geht. Das Team kann sich besser auf dynamische, soziale Prozesse einstellen.

Zusammenfassend kann man sagen, dass User Stories ein leichtgewichtiges Mittel zur Planung und Umsetzung einer Projektarbeit sind, die sich sowohl in der Domäne des Kunden, als auch in der Domäne der Entwickler befindet.

5.3 Testfirstansatz

Der Testfirstansatz beschreibt einen Programmierstil, der seine Priorität in der Qualitätssicherung sieht. Das Problem bei der Erstellung umfangreicher Software ist es sie fehlerfrei und kompakt zu halten. Mit jeder zusätzlichen Zeile Quellcode vergrößert sich das Fehlerrisiko und der Umfang des später zu wartenden Programms. Das hat zur Folge, dass bereits bei kleinen, auf den ersten Blick übersichtlich erscheinenden Programmen der Aufwand sie fehlerfrei zu halten, sehr groß werden kann. Auch kann das Hinzufügen neuer Funktionen zu Fehlern im bereits für fehlerfrei befundenen Code führen.[XP]

Die Wartungsaufgaben können sich durch das Phänomen des „Aufplustern“ von Quellcode noch vergrößern, indem Programmierer dazu verleitet werden, zusätzlichen, aber für die zu programmierende Aufgabe irrelevanten, Code zu produzieren. Auch Programmierer sollen sich kurz fassen.

Der Testfirstansatz versucht diesen Problemen zu begegnen, indem vor der eigentlichen Programmierarbeit kleine Testprogramme geschrieben werden, die den zu produzierenden Code automatisch testen können. Im Kontext der objektorientierten Softwareentwicklung bedeutet das, dass zu jeder geschriebenen

Klasse eine spezielle Testklasse gehört, welche die einzelnen Methoden der Klasse testet.

Indem der Test vor dem Schreiben des eigentlichen Codes erzeugt wird, bekommt der Programmierer ein zusätzliches Designwerkzeug in die Hand. Der vor dem Programmieren einer Methode geschriebene Test, beschreibt Funktionsweise und Verhalten der Methode.

Während seiner anschließenden Arbeit kann der Programmierer ständig automatisch und zu jeder Zeit seinen produzierten Code testen, was bedeutet, dass die Arbeit sofort nach dem ersten erfolgreichen Test beendet ist. Dadurch kann aufgeplusterter Code fast vollständig vermieden werden. Auch wird man die einmal geschriebenen Tests nicht verwerfen, sondern sammeln, so dass man mit der Zeit eine umfangreiche Testbibliothek in der Hand hat, mit der die Qualität des Produkts gesichert werden kann.

Durch diese Menge an Tests lassen sich schnell Probleme finden, die nach der Integrationen neuer Funktionen in das bestehende Programm, an einer völlig anderen Stelle entstehen können. Die Integrität des gesamten Systems lässt sich so sicherstellen.

5.4 JUnit

JUnit ist ein Testframework für Java um automatisierte Tests schnell und einfach zu erstellen und durchzuführen. Es folgt einer einfachen Ampelsemantik um erfolgreiche oder fehlgeschlagene Tests anzuzeigen und kann mehrere Tests zu Unittests zusammenfassen. Somit lassen sich ganze Programmmodule auf einmal testen.

Die Programmierung eines Tests ist denkbar einfach. Eine neue Testklasse wird von der bestehenden Klasse `Test` abgeleitet. Die Klasse `Test` besitzt verschiedene Methoden zum Erschaffen und Aufräumen von Testumgebungen, die überschrieben werden können. Einzelne Tests werden durch neue Methoden beschrieben, die alle mit dem Wort *test* anfangen und vom Testframework automatisch erkannt und ausgeführt werden. Beim Testen helfen verschiedene `assert` Methoden, mit denen Systemzustände abgefragt werden können und deren Fehlschlag auch einen Fehlschlag des gesamten Tests markiert. [JU]

6. Planung

6.1 Übersicht

Dieses Kapitel befasst sich mit der Planung dieser Diplomarbeit und wie diese mit ausgesuchten Mitteln des Extreme Programing durchgeführt wurde.

Nachfolgend werden die geplanten User Stories mit ihren Tasks aufgelistet. Jede User Story besitzt eine Priorität für den Kunden, eine Gewichtung für das Gesamtprojekt und eine Risikobewertung des Entwicklers. Alle Werte gehen von 1 (= niedrig) bis 5 (=hoch).

An die Beschreibung jeder Task einer User Story fügt sich eine Liste der wichtigsten Klassen und Interfaces an, die zur Realisierung der jeweiligen Task erstellt wurden. An dieser Stelle soll auf diese Klassen nicht weiter eingegangen werden, sie dienen nur der Übersicht. Im nachfolgenden Kapitel zur neuen Architektur werden die Klassen im Gesamtkontext näher beschrieben.

In Form von Anmerkungen wird auf jede User Story noch einmal einzeln eingegangen und ihre Position im Projekt und ihre Bedeutung für die neue Architektur näher erläutert.

Auffallend ist die durchweg hohe Priorität und Gewichtung der User Stories. Das hat seinen Ursprung in der Überlegung, dass alle User Stories Teil eines Gesamtpaketes sind, das nur schwer auf einzelne Teile verzichten kann.

6.2 Service-Framework

Service Framework		
Priorität: 5	Risiko: 2	Gewichtung: 5
Beschreibung	Entwicklung eines Frameworks, das Services zur Verfügung stellt, die in beliebiger Abfolge untereinander angeordnet werden können, um eine Aufgabe zu erfüllen.	
Erfolgskriterium	Mehrere Services sollen zu einer Service-Kette zusammen gestellt werden, die in einem Auftrag (Job) nacheinander abgearbeitet werden können.	
Tasks		
Services	Basis für das Serviceframework aus einer erweiterbaren Serviceklasse die zur Erfüllung einer Task benutzt wird.	
	org.dotplot.core.services.IService org.dotplot.core.services.IFrameworkContext org.dotplot.core.services.IContext org.dotplot.core.services.IServiceRegistry	
Nebenläufigkeit	Einzelne Services sollen ihre Funktionalität in Teilaufgaben unterteilen können, die unabhängig von einander ausführbar sein sollen und am Ende zu einem Gesamtergebnis wieder zusammengefasst werden können. Diese Funktionalität soll abhängig von den Aufgaben des Service zur Verfügung gestellt werden.	
	org.dotplot.core.services.ITask org.dotplot.core.services.ITaskPart org.dotplot.core.services.ITaskProcessor org.dotplot.core.services.ITaskResultMarshaler org.dotplot.core.services.IResources	
Service HotSpots	Die Services mit Hotspots versehen, um Punkte zu definieren, an denen sie erweitert werden können	
	org.dotplot.core.services.Extention org.dotplot.core.services.IServiceHotspot org.dotplot.core.services.IExtentionActivator	
Jobs	Eine Job Klasse übernimmt die Ausführung der Services	
	org.dotplot.core.services.IJob	
Anmerkungen		
Ziel dieser User Story war es die Basis der neuen Architektur zu legen. Die hier geschaffenen Klassen wirken sich in alle Bereiche von Matrix4.plot aus. Technisch gesehen wurden keine Probleme erwartet, weshalb das Risiko für diese User Story als gering eingestuft wurde.		

6.3 Pluginaufsatz für das Service Framework

<i>Pluginaufsatz für das Service Framework</i>		
Priorität: 5	Risiko: 2	Gewichtung: 4
Beschreibung	Einen Pluginmechanismus für das Service Framework auf XML-Basis.	
Erfolgskriterium	Die im System verfügbaren Services sollen durch die XML-Datei beeinflussbar sein.	
<i>Tasks</i>		
Technical Spike: XML-Funktionalität von Java testen	Prüfen, ob sich die von Java unterstützten XML-Funktionen für die Plugin-Architektur verwenden lassen.	
	keine	
Technical Spike: Java-Reflection testen	Prüfen, wie der Reflectionmechanismus von Java für den Pluginmechanismus eingesetzt werden kann.	
	org.dotplot.util.JarFileClassLoader	
Pluginverwaltung	Ein System zur Verwaltung von Plugins muss geschaffen werden. Es soll eine Versionierungskontrolle und eine Abhängigkeitskontrolle beinhalten.	
	org.dotplot.core.plugins.IPlugin org.dotplot.core.plugins.IPluginRegistry org.dotplot.core.plugins.PluginContext org.dotplot.core.plugins.Version	
Neue Services	Über den Pluginmechanismus sollen neue Services dem System hinzugefügt werden können.	
	org.dotplot.core.plugins.PlugableService org.dotplot.core.plugins.PluginHotspot	
Neue Jobs	Über den Pluginmechanismus sollen neue Jobs dem System hinzugefügt werden können. Alternativ sollen Batchjobs in der plugin.xml definiert werden können.	
	org.dotplot.core.plugins.IJobRegistry org.dotplot.core.plugins.BatchJob	
Extentions	Serviceextentions sollen für bestehende und neu hinzugekommene Services definiert werden können.	
	org.dotplot.core.plugins.IExtentionFactory	

Pluginloader	Ein spezieller Service zum laden der Plugins.
	org.dotplot.core.plugins.PluginJarFile org.dotplot.core.plugins.PluginLoadingService org.dotplot.core.plugins.PluginIntegrationService org.dotplot.core.plugins.InitializerService
Anmerkungen	
<p>Die beiden Technical Spikes⁶ wurden durchgeführt um zu testen wie und ob sich Java interne Mechanismen einsetzen lassen. Sie sind beide erfolgreich gewesen, weshalb mit den dort gewonnenen Erkenntnissen gleich weitergearbeitet werden konnte.</p> <p>Ansonsten war das Ziel dieser User Story, die Erweiterbarkeit der neuen Architektur zu realisieren. Auch hier war das technische Risiko gering, und die Gewichtung für das Projekt hoch.</p>	

⁶ Technical Spike: Die Untersuchung einer Technologie auf ihre Relevanz für das Projekt. Im allgemeinen werden kleine Testprogramme geschrieben, die prüfen ob die Technologie das leistet was man sich von ihr verspricht.

6.4 Adaption des Plugin-Framework für Matrix4.plot

Adaption des Plugin Framework für Matrix4.plot		
Priorität: 4	Risiko: 3	Gewichtung: 4
Beschreibung	Das Service-Framework soll für Matrix4.plot adaptiert werden. Klar definierte Erweiterungspunkte sollen das System flexibel erweiterbar machen (neue Scanner, Filter, Dotplot-darstellungen, F-Matrix, Q-Matrix).	
Erfolgskriterium	Umstieg auf die neue Architektur. Das System soll eine Grundfunktionalität besitzen, um ohne großen Aufwand Dotplots zu erstellen. Diese Grundfunktionalität soll durch Plugins auf die Funktionalität der bestehenden Version von Matrix4.plot erweitert werden, so dass sich für den Anwender keinerlei Unterschiede feststellen lassen.	
Tasks		
GUI zur Pluginverwaltung	GUI um die installierten Plugins zu verwalten. Die GUI soll im Preferences-Menü von Eclipse zu finden sein.	
	org.dotplot.eclipse.preference.PluginOverviewPage org.dotplot.eclipse.preference.PluginOverviewGui	
Tokenizer-Service	Ein Service, der aus einer Plot-Quelle einen Tokenstrom macht.	
	org.dotplot.tokenizer.service.TokenizerService org.dotplot.tokenizer.service.ITokenizerConfiguration	
Tokenfilter-Service	Ein Service der Tokenfilter zur Verfügung stellt und einsetzt, um Tokenströme zu modifizieren.	
	org.dotplot.tokenizer.filter.FilterService org.dotplot.tokenizer.filter.IFilterConfiguration	
Konvertierungs Service	Neue Datenquellen und Formate sollen eingefügt und über Konverter in andere Formate überführt werden können.	
	org.dotplot.tokenizer.converter.ConverterService org.dotplot.tokenizer.converter.IConverterConfiguration	
GUI Anpassung	Anpassung der bestehenden Konfigurations GUI an die neuen Erfordernisse des Service-Frameworks.	
	org.dotplot.tokenizer.filter.ui.IFilterUI org.dotplot.tokenizer.filter.ui.SelectFilterUI	
F-Matrix Service	Ein Service für die F-Matrix.	
	org.dotplot.fmatrix.FMatrixService org.dotplot.fmatrix.IFMatrixConfiguration	

Q-Image Service	Ein Service für das Q-Image.
	org.dotplot.image.QImageService org.dotplot.image.IQImageConfiguration
Anmerkungen	
Diese User Story war die erste Bewährungsprobe für das geschaffene Framework und beinhaltete das größte Risiko. Wäre sie nicht umsetzbar gewesen, hätte die Arbeit von vorne beginnen müssen. Auch handelt es sich um das Herzstück der Restrukturierung und gab der Paketstruktur von Matrix4.plot ein neues Aussehen.	

6.5 Adaption der GUI auf des Plugin-Framework

Adaption der GUI auf des Plugin Framework		
Priorität: 3	Risiko: 3	Gewichtung: 4
Beschreibung	Die GUI des RPC bzw. des Eclipse-Plugins soll mit Hilfe des Plugin Frameworks erweitert werden können.	
Erfolgskriterium	Neue GUI-Elemente können durch Plugins hinzugefügt werden.	
Tasks		
GUI Service	Eigener Service um die Eclipse-GUI mit Dotplot-Plugins zu erweitern.	
	org.dotplot.core.IGuiService org.dotplot.eclipse.EclipseUIService	
Jobs als Actions	Jobs sollen als Actions eingefügt werden können.	
	org.dotplot.eclipse.actions.JobAction	
Anmerkungen		
Diese User Story ist der erste Schritt, um die Abhängigkeit zwischen dem Eclipse Pluginmechanismus und dem Matrix4.plot Pluginmechanismus zu durchbrechen. Vorher mussten alle GUI Elemente über Eclipse in das Programm eingebunden werden. Nach Abschluss dieser User Story ist das nicht mehr für Menüeinträge nötig. Sie können nun über Einstellungen in der <i>dotplotplugin.xml</i> Datei des Plugins hinzugefügt werden. Außerdem werden über den GuiService Aufrufe zur Nachrichtenweitergabe an den Anwender vereinheitlicht.		

7. Umsetzung

7.1 Konzeptionelle Änderungen

Ausgehend von der bestehenden Architektur fallen schnell zwei Probleme auf:

- Ein Dotplot kann nur aus Dateien generiert werden.
- Die Möglichkeiten der Eclipse Plattform werden nicht voll ausgeschöpft.

Das erste Problem wird deutlich in der Überlegung, wie einschränkend das Konzept ist. Das Plotten einer Netzquelle, wie eine online verfügbare HTML-Seite, ist nur über Umwegen möglich. Die HTML-Seite muss erst als Datei lokal gespeichert werden, bevor sie geplottet werden kann.

Das zweite Problem wird klar, wenn man sich Gedanken darüber macht, dass die Eclipse-Plattform neben ihrer Eigenschaft als Laufzeitumgebung auch eine Reihe von nützlichen Plugins zur Verfügung stellt, die es ermöglichen würden die Arbeit mit `Matrix4.plot` noch benutzerfreundlicher zu machen. So kann eine kontextsensitive Onlinehilfe integriert oder eine automatische Update-Funktion eingebaut werden. Aber auch einfachere Dienste der Eclipse Plattform, wie die Datei- und Konfigurationsverwaltung, sind in der bestehenden Architektur nicht vorgesehen.

Beide Probleme sind konzeptioneller Art und erstrecken sich durch alle Bereiche des Programms. Aus diesem Grund müssen sie vor der weiteren Arbeit gelöst werden.

Plotquellen

Wie bei vielen objektorientierten Programmen hat die Domäne und die ihr innewohnende Nomenklatur großen Einfluss auf das entstehende Programm. Daher ist es nicht verwunderlich, dass während der ersten Projektarbeit an `Matrix4.plot` die Klasse `File` als Quelle von Dotplots verwendet wurde. Die Klasse `File` repräsentiert eine physikalische Datei auf dem ausführenden System der JavaLaufzeitumgebung und bietet in ihrer Funktionalität den Kern der ersten Aufgabe des Projekts zu lösen: Erstellung eines Dotplots aus einer Datei.

Im Nachhinein erwies sich diese Designentscheidung als unflexibel und daher ungeeignet.

Ein besserer Entwurf ist die Verwendung einer Plotquelle, die verschiedene Ausprägungen haben kann, wie Datei oder Datenbankverbindung. Durch diese Änderung im Konzept wird es möglich, flexibler auf zukünftige Verwendungen eingehen zu können.

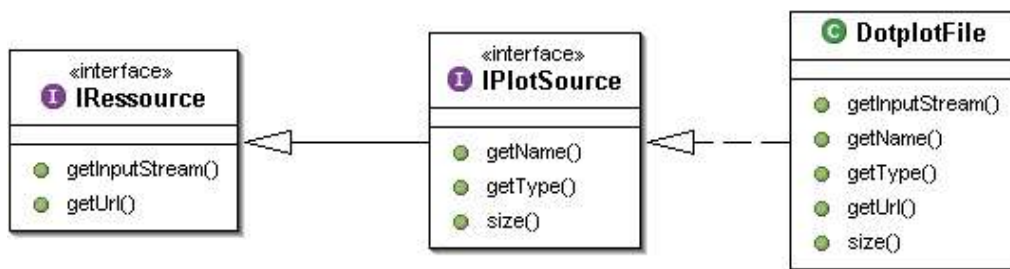


Diagramm 3 - Konzept der Plotquellen

Das Interface `IPlotSource` ist der Basistyp aller möglichen Plotquellen und kann zukünftig durch weitere Plotquellen implementiert werden. Das Interface `IReSSource` steht für Objekte die eine vom System verwendbare Ressource darstellen. Das ist im Wesentlichen etwas, das lokalisierbar (URL) und lesbar (InputStream) ist.

Typisierung

Die Verwendung von Plotquellen als Oberbegriff macht es möglich diese auch zu typisieren, und zwar umfangreicher als das mit Dateien der Fall gewesen wäre. Durch den Quelltyp lassen sich mit dem Mittel der Vererbung Typhierarchien aufbauen auf die das Programm reagieren kann. So wird es möglich Tokenizer auf Basis des Typs der ausgewählten Plottquelle auszuwählen, um zu verhindern dass Tokenizer auf die falsche Art von Informationen angewandt werden.

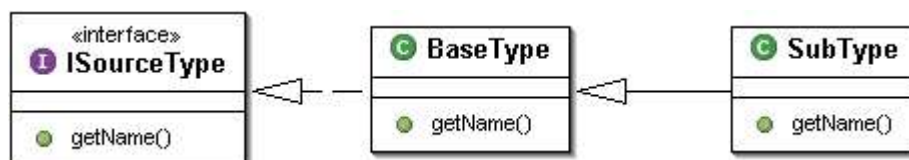


Diagramm 4 - Konzept der Typisierung

Die Schnittstelle `ISourceType` ist der Basistyp und `BaseType` ist eine Standardimplementation der Schnittstelle. `BaseType` dient als Wurzel der Typhierarchie und symbolisiert etwas, das sich plotten lässt, bzw. als Quelle für einen Dotplot dienen kann.

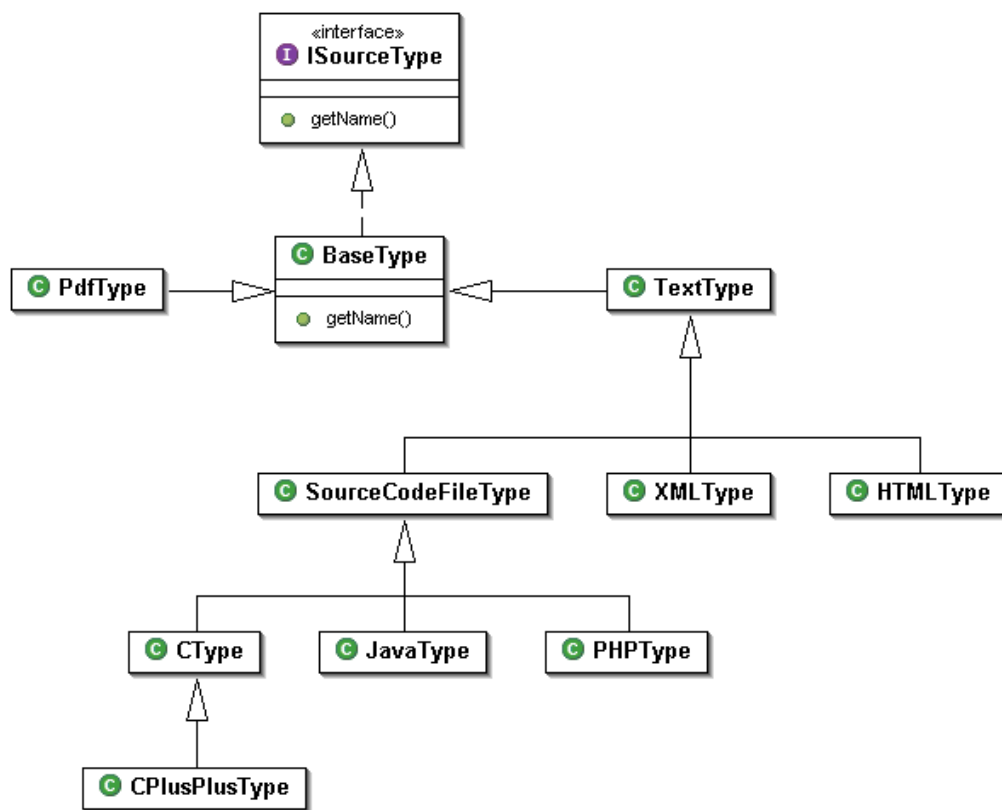


Diagramm 5 - Typhierarchie der neuen Architektur

Durch Erweitern der Klasse `BaseType` lassen sich Spezifizierungen realisieren, durch die sich die Typen in Vererbungsgruppen zusammenfassen lassen. So werden die Typen von Java und C++ Dateien als Programmiersprachen zusammengefasst, die wiederum zu der Gruppe der Textdateien gehören.

Durch diese Vererbungsstruktur lassen sich gezielt Transformationen für bestimmte Plottquellen realisieren. Zum Beispiel Tokenizer die für alle Quelldateien (`SourceCodeFileType`) gelten, oder Filter die nur auf XML-Dateien (`XMLType`) anwendbar sind. Durch eine Spezialisierung innerhalb der Typhierarchie werden die möglichen Transformationen erweitert. Transformationen auf einer C Datei (`CType`) kann immer auch auf einer C++ Datei ausgeführt werden, umgekehrt jedoch nicht.

In diesem Hintergrund realisiert ein Konverter einen Sprung innerhalb der Typhierarchie. Eine PDF-Datei kann dadurch zu einer Textdatei werden und sich in einem ganz anderen Ast der Typhierarchie wiederfinden. Dort können neue Transformationen auf ihr angewandt werden, die vorher nicht möglich waren.

Konfigurationsmanagement

Bereits in der ersten Version von Matrix4.plot fällt auf, dass die verwendete Plotkonfiguration nicht gespeichert wird und auch nicht in die nächste Sitzung übernommen werden kann. Es liegt nahe, dies von Eclipse erledigen zu lassen. Das dabei auftauchende Problem ist aber, dass es zwangsläufig zu einer sehr engen

Verzahnung zwischen Eclipse und Matrix4.plot kommt. In seiner gegenwärtigen Form, wäre es möglich, mit nur wenigen Anpassungen am eigentlichen Programmcode, Matrix4.plot auf eine andere Plattform zu emigrieren. Das wäre nicht mehr möglich, wenn die Stellen in denen sich der Programmcode auf die Eclipsekonfiguration bezieht in die Höhe schnellen.

Daher ist es eine gute Idee die Eclipsekonfiguration durch ein eigenes Konfigurationsmanagement zu verbergen, das intern auf die Eclipsekonfiguration zugreift, um seine Aufgabe zu erfüllen.

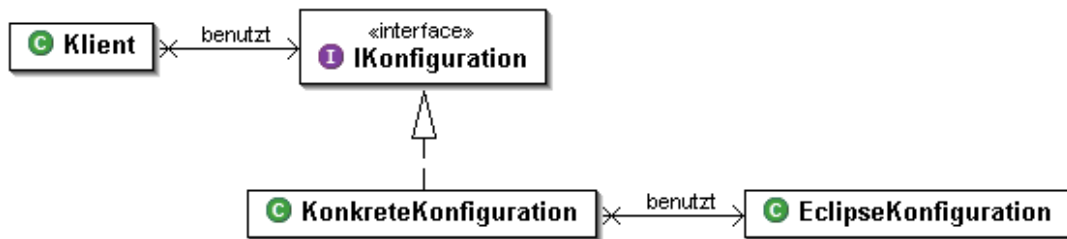


Diagramm 6 - Konzept des Konfigurationsmanagements

7.2 Das Service-Framework

Übersicht

Da sich die bestehende Architektur an *Pipes & Filters* orientiert, wäre es nur sehr schwer diese Architektur zu verändern. Das liegt unter anderem daran, dass *Pipes & Filters* dem natürlichen Transformationsfluss der Quellinformationen hin zum fertigen Dotplot sehr ähnlich ist. Aus diesem Grund ist es besser auch die neue Architektur an diesem Konzept zu orientieren, um den bestehenden Code so wenig wie möglich anpassen zu müssen.

`Services` bilden die Grundlage der neuen Architektur. Der Name ist zunächst irreführend, da er schnell mit normalen Diensten des Systems verwechselt werden kann. Im Kontext von `Matrix4.plot` ist ein `Service` eine gekapselte Ausführungseinheit für eine bestimmte Aufgabe. Demnach kann er ausgeführt werden und produziert dabei ein Ergebnis auf Basis einer Eingabe.

Seine Eingabedaten erhält ein `Service` in Form eines `WorkingContext`, der alle benötigten Eingabedaten für den `Service` bereitstellt. Nach seiner Ausführung produziert der `Service` einen `ResultContext` der alle Ausgabedaten enthält.

Wie in *Pipes & Filters* können `Services` hintereinander geschaltet werden und bilden dabei Ausführungsketten. Entsprechend dieser Architektur ist ein `Service` einem `Filter` gleichzusetzen. Der `ResultContext` bildet dabei den `WorkingContext` des in der Kette nachfolgenden `Service` und ist einer `Pipe` gleichzusetzen.

Um die Kompatibilität der `Services` innerhalb der Ausführungskette sicherzustellen, stellt jeder `Service` eine Methode zur Verfügung, um die Kompatibilität des `WorkingContexts` festzustellen und eine Methode, um die Klasse des `ResultContext` zu erfahren.

Zusätzlich zu seinem `WorkingContext` arbeitet ein `Service` auch in einem `FrameworkContext`, der dem `Service` erlaubt auf allgemeine Dienste des Systems zuzugreifen.

Ein `Service` ist im System als persistentes Objekt vorhanden, das in einer `ServiceRegistry` mit einer eindeutigen ID registriert ist. Die `ServiceRegistry` ist Teil des `FrameworkContext`. Auf diese Weise kann ein `Service` von jedem Objekt gefunden werden, das den `FrameworkContext` kennt.

Seine Aufgabe erledigt ein `Service` mit Hilfe einer `Task`, die er mit dem Fabrikmethode Entwicklungsmuster erzeugt. Eine `Task` übernimmt die eigentliche Ausführung der Aufgabe des `Service`. Der `Service` konfiguriert die `Task` mit Hilfe der ihm zur Verfügung stehenden Eingabedaten, so dass alle Vorbedingungen zur Ausführung der `Task` erfüllt sind.

Eine `Task` besteht wiederum aus einzelnen `TaskParts`, welche die Aufgabe der `Task` in Teilaufgaben unterteilen und einzeln bearbeitet werden können. `Tasks`,

die ein solches Vorgehen unterstützen, sind teilbar und die einzelnen `TaskParts` können völlig unabhängig vom übrigen System bearbeitet werden. Dadurch wird eine Parallelverarbeitung möglich.

Um die Art und Weise wie man eine `Task` bearbeitet flexibel zu machen, wird das *Strategie* Muster angewandt. Ein `TaskProcessor` bestimmt wie eine `Task` bearbeitet wird. Ein besonderes Augenmerk liegt hier auf der Reihenfolge in der die einzelnen `TaskParts` bearbeitet werden.

Um die Ergebnisse der einzelnen `TaskParts` zu einem Gesamtergebnis zusammenzusetzen wird sich ebenfalls des *Strategie* Musters bedient. Ein `TaskResultMarshaller` wird dem `TaskProcessor` zugewiesen und übernimmt die Aufgabe das Gesamtergebnis zu erzeugen.

Die Erweiterbarkeit eines `Service` wird mit `Hotspots` realisiert. Ein `Hotspot` stellt eine verallgemeinerte Schnittstelle zur Verfügung, über die neue Funktionen in den `Service` integriert werden können.

Ein `Hotspot` wird definiert indem er einem `Service` zusammen mit einer `ID` und einem `Class` Objekt übergeben wird, das den Typ der Erweiterungen festlegt. Beim Hinzufügen einer Erweiterung prüft der `Hotspot` automatisch, ob die Erweiterung zum angegebenen `Class` Objekt passt.

Eine Erweiterung ist ein Objekt der Klasse `Extention` und verwaltet ein Tupel bestehend aus einem Objekt der Klasse `ExtentionActivator` und dem eigentlichen Objekt der Erweiterung. Ein `ExtentionActivator` bietet die Möglichkeit eine Erweiterung gezielt zu aktivieren bzw. zu deaktivieren.

Zusätzlich zu diesen beiden Objekten verwaltet ein `Extention` Objekt auch noch beliebig viele Parameter zu einer Erweiterung, wodurch sich die Verwendung der Erweiterung innerhalb des `Service` weiter spezifizieren und anpassen lässt.

Die Ausführung eines `Service` oder einer *Servicekette*, wird durch einen `Job` angestoßen. Tatsächlich sind `Jobs` die einzigen Objekte, die vom Anwender direkt über seine Eingabe ausgeführt werden können. Sie bedienen sich dafür des Entwicklungsmusters *Befehl* um diese Ausführung zu kapseln.

Auch wenn `Jobs` hauptsächlich `Services` und *Serviceketten* ausführen, so sind sie davon trotzdem unabhängig und sind bei dem was sie ausführen nicht festgelegt. Im Rahmen des *Pipes & Filters* Musters übernimmt ein `Job` die Aufgabe die `Services` nacheinander auszuführen und die Weitergabe der Daten in Form von `WorkingContext` und `ResultContext` sicherzustellen.

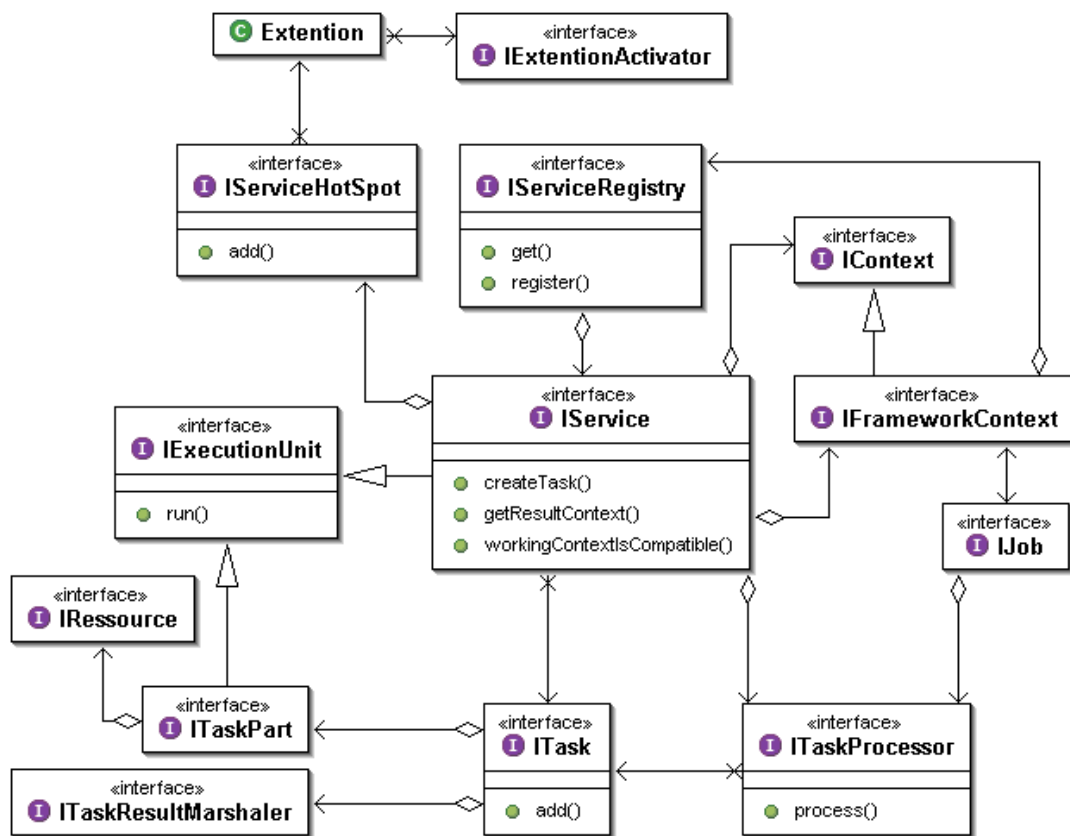


Diagramm 7 - Übersicht Service Framework

Muster: *Befehl*

Quelle: [GoF]

Zweck

Kapselt einen Befehl als ein Objekt. Dies ermöglicht es, Klienten mit verschiedenen Anfragen zu parametrieren, Operationen in eine Queue zu stellen, ein Logbuch zu führen und Operationen rückgängig zu machen.

Struktur

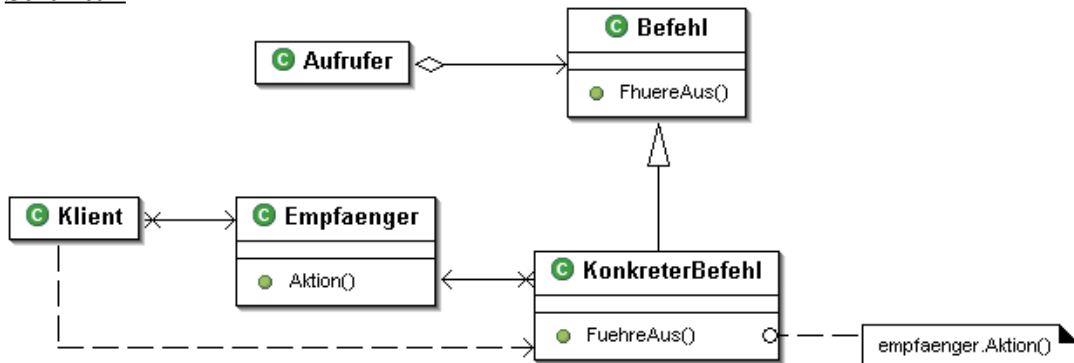


Diagramm 8 - Struktur des Musters Befehl

Teilnehmer

- **Befehl**
 - deklariert eine Schnittstelle zum Ausführen einer Operation
- **KonkreterBefehl**
 - definiert die Anbindung eines Empfängers an eine Aktion.
 - implementiert `FuehreAus()` durch Aufrufen der entsprechenden Operationen beim Empfänger.
- **Klient**
 - erzeugt ein `KonkreterBefehl`-Objekt und übergibt ihn dem Empfänger.
- **Aufrufer**
 - befiehlt dem `Befehl`-Objekt die Anfrage auszuführen.
- **Empfänger**
 - weiß, wie die an die Ausführung einer Anfrage gebundenen Operationen auszuführen sind. Jede Klasse kann ein Empfänger sein.

Befehl in der Anwendung

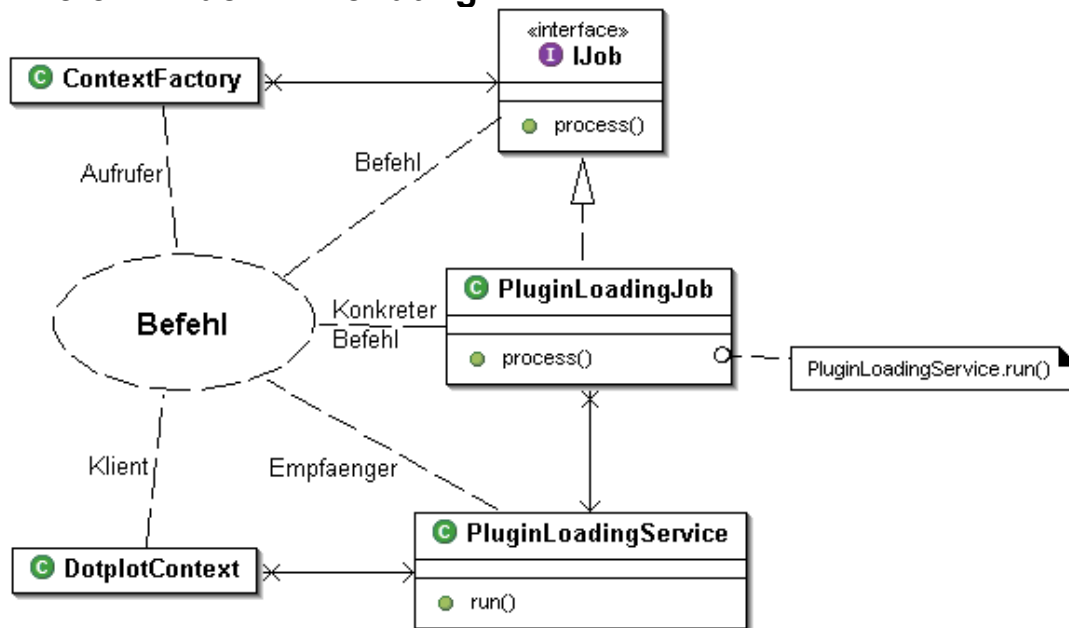


Diagramm 9 - Anwendung von Befehl

Die ContextFactory benutzt beim erzeugen des DotplotContext den PluginLoadingJob. Der PluginLoadingJob ruft den PluginLoadingService auf, um die Plugins zu laden, die in den DotplotContext integriert werden.

Auf die für dieses Beispiel verwendeten Klassen, wird in den nachfolgenden Abschnitten näher eingegangen.

Muster: Strategie

Quelle: [GoF]

Zweck

Definiere eine Familie von Algorithmen, kapsle jeden einzelnen Algorithmus und mache sie austauschbar. Das Strategiemuster ermöglicht es, den Algorithmus unabhängig von ihn nutzenden Klienten zu variieren.

Struktur

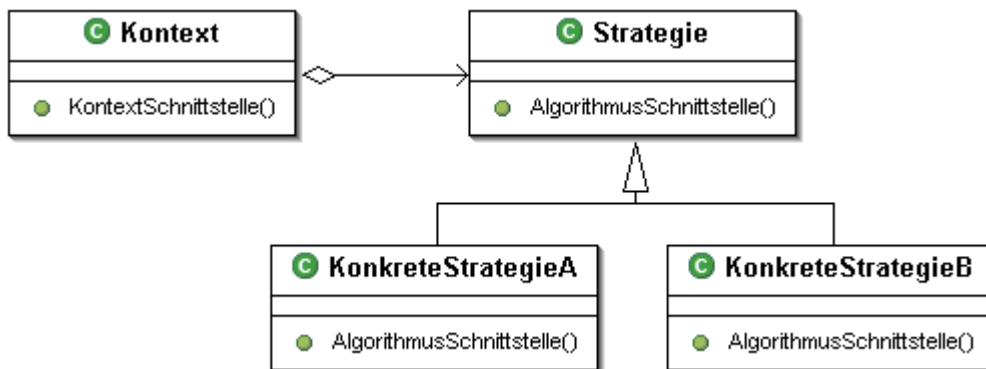


Diagramm 10 - Struktur des Musters Strategie

Teilnehmer

- **Strategie**
 - deklariert eine Schnittstelle, die von allen unterstützten Algorithmen angeboten wird. Das **Kontext**-Objekt verwendet diese Schnittstelle, um den durch eine **KonkreteStrategie** definierten Algorithmus aufzurufen.
- **KonkreteStrategie**
 - implementiert den Algorithmus unter Verwendung der Strategie-Schnittstelle.
- **Kontext**
 - wird mit einem **KonkreteStrategie**-Objekt konfiguriert.
 - verwaltet eine Referenz auf ein **Strategie**-Objekt.
 - kann eine Schnittstelle definieren, die **Strategie**-Objekten den Zugriff auf seine Daten ermöglicht.

Strategie in der Anwendung

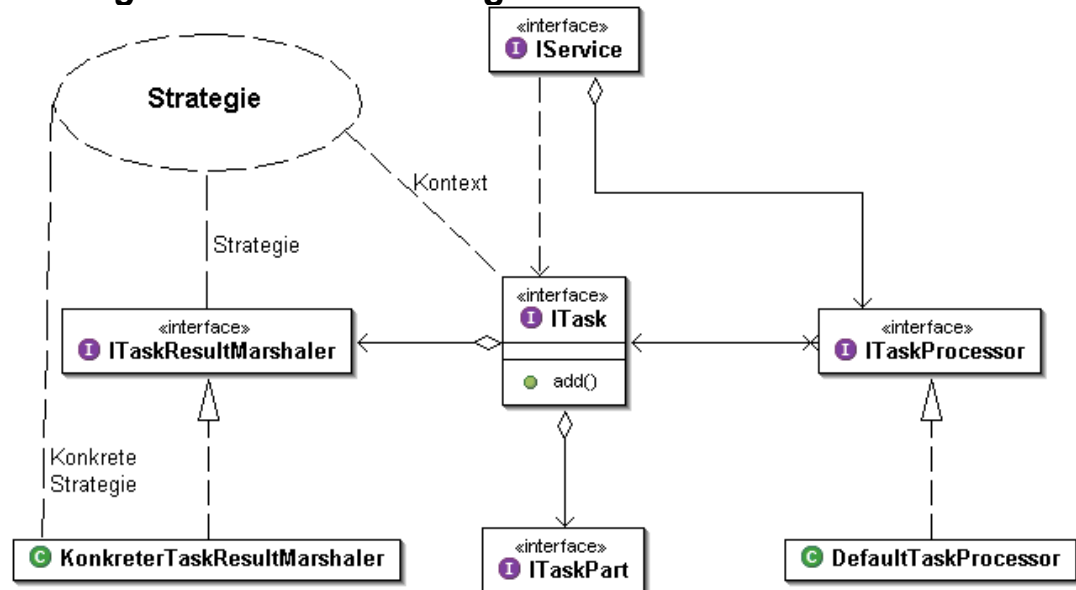
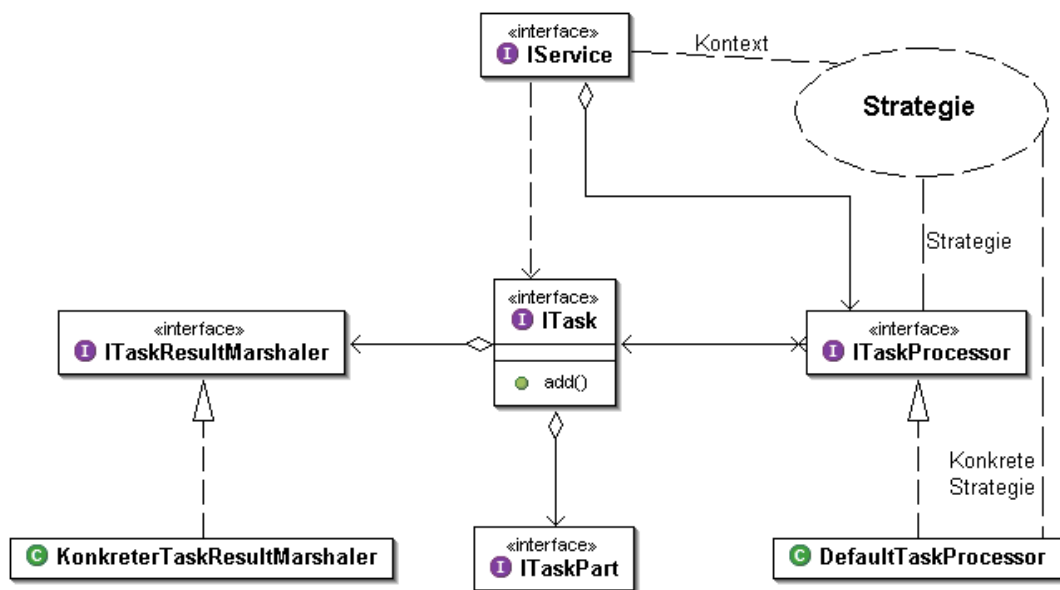


Diagramm 11 - Verwendung von Strategie 1

Hier ist zu erkennen, wie *Strategie* angewendet wird, um das Zusammenfügen des Task Resultats flexibel zu gestalten. Das Resultat eines Task hängt von der Implementation der Task selbst ab und kann vom Frameworkentwickler nicht vorhergesehen werden.

TaskParts sollen unabhängig voneinander verarbeitbar sein. Das bedeutet besonders, dass die Reihenfolge der Verarbeitung keinen Einfluss auf das Ergebnis haben darf. Daher ist dem Serviceentwickler unmöglich die Reihenfolge der Verarbeitung vorherzusehen, weshalb die Ergebnisse der einzelnen Tasks vom TaskProcessor gesammelt werden. Nach Abschluss des letzten TaskPart, übergibt der TaskProcessor die gesammelten Ergebnisse dem TaskResultMarshaller, der aus ihnen das Gesamtergebnis der Task erstellt.



Mit Hilfe von *Strategie* wird auch festgelegt wie eine Task bearbeitet wird. Der Algorithmus wie der Service seine Task bearbeitet ist nicht fest eingebaut, um die Flexibilität zu bewahren, die Verarbeitung parallel oder verteilt durchführen zu können.

Auch kann die Effektivität eines Verarbeitungsalgorithmus von der verarbeiteten Aufgabe abhängig sein und mit *Strategie* kann gut darauf eingegangen werden.

Die Verarbeitungsalgorithmen werden als TaskProcessoren dem Service zugewiesen. Wird der Service ausgeführt benutzt er den TaskProcessor, um seine Task zu bearbeiten.

Muster: *Fabrikmethode*

Quelle: [Gof]

Zweck

Definiere eine Klassenschnittstelle mit Operationen zum Erzeugen eines Objekts, aber lasse Unterklassen entscheiden, von welcher Klasse das zu erzeugende Objekt ist. Fabrikmethoden ermöglichen es einer Klasse, die Erzeugung von Objekten an Unterklassen zu delegieren.

Struktur

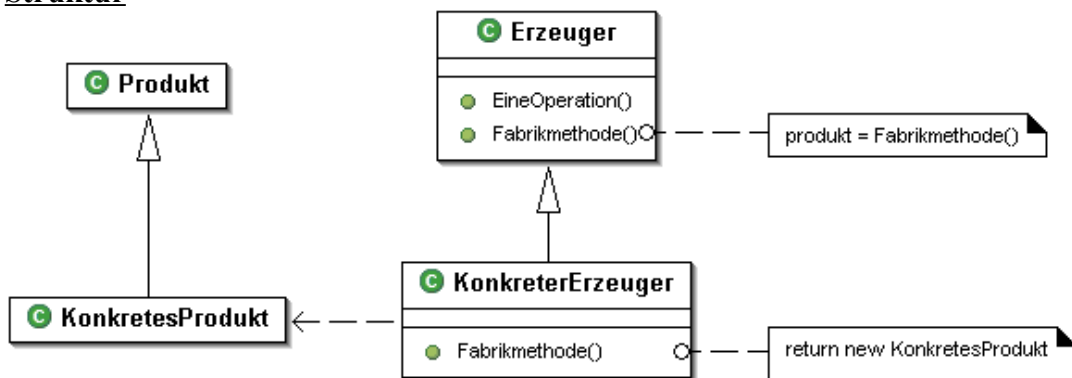


Diagramm 13 - Struktur des Musters Fabrikmethode

Teilnehmer

- **Produkt**
 - definiert die Klasse des von der Fabrikmethode erzeugten Objekts.
- **KonkreterProdukt**
 - implementiert die Produktschnittstelle.
- **Erzeuger**
 - deklariert die `Fabrikmethode`, die ein Objekt des Typs `Produkt` zurückgibt. Der Erzeuger kann möglicherweise eine Standardimplementierung der `Fabrikmethode` definieren, die ein vordefiniertes `KonkreterProdukt`Objekt erzeugt.
 - kann die `Fabrikmethode` aufrufen, um ein Produktobjekt zu erzeugen.
- **KonkreterErzeuger**
 - überschreibt die `Fabrikmethode`, so dass sie ein Exemplar von `KonkreterProdukt` zurückgibt.

Fabrikmethode in der Anwendung

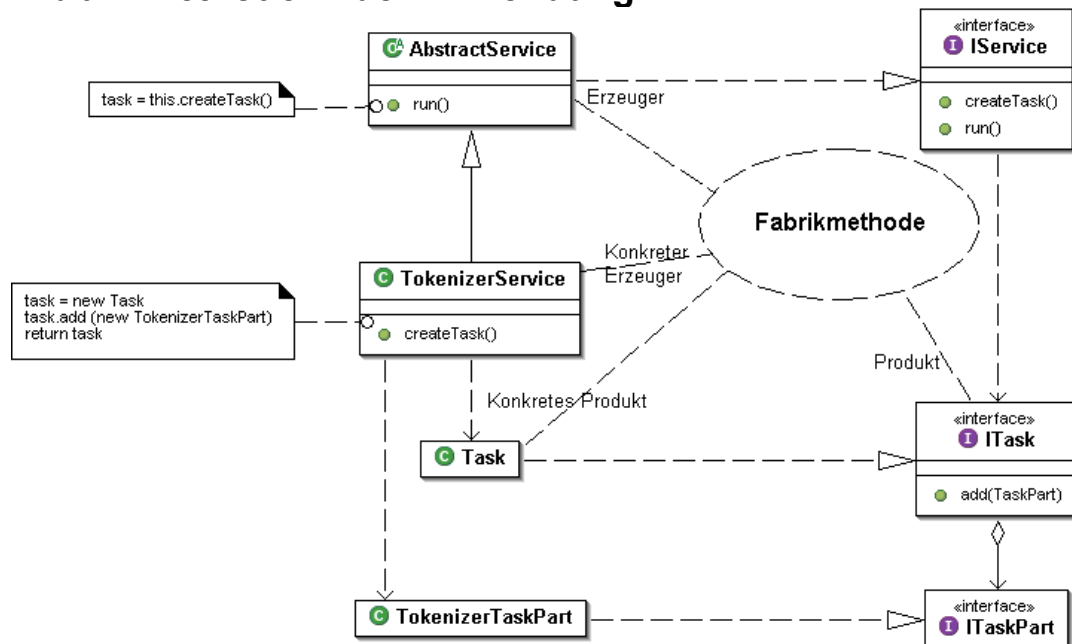


Diagramm 14 - Fabrikmethode in der Anwendung

Mit Hilfe von *Fabrikmethode* legt der Entwickler der Klasse **TokenizerService** fest, wie die **Task** des Service erzeugt wird. Die Verwendung des erzeugten **Task** Objekts ist zusammen mit dem Aufruf von `createTask()` in der Klasse **AbstractService** vom Framework vorgegeben.

Der Entwickler kümmert sich also nur noch um die Erzeugung des Produkts und braucht sich um seine richtige Verwendung im Rahmen des Frameworks nicht mehr zu kümmern.

Muster: *Registry*

Quelle: [PEAA]

Zweck

Ein wohlbekanntes Objekt, das andere Objekte benutzen können, um gemeinsame Objekte und Dienste zu finden.

Struktur

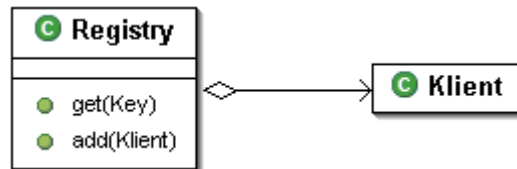


Diagramm 15 - Struktur des Musters Registry

Teilnehmer

- **Registry**
 - verwaltet Beziehungen zwischen Klienten und eindeutigen Schlüsseln.
 - liefert nach Angabe des Schlüssels den richtigen Klient.
 - meist als Singleton⁷ implementiert.
- **Klient**
 - kann durch die Registry gefunden werden.

⁷ Eine Klasse von der zu jeder Zeit nur ein Objekt existiert. [Gof]

Registry in der Anwendung

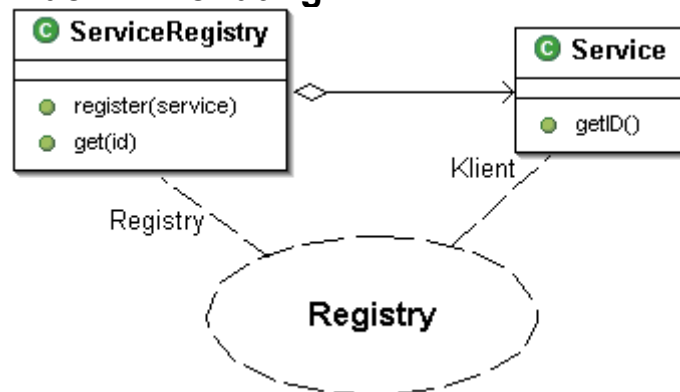


Diagramm 16 - Registry in der Anwendung

Registry ist ein viel verwendetes Muster. An diesem Beispiel kann man sehen, dass Services in der ServiceRegistry gespeichert werden. Als Schlüssel dient die id des Service.

7.3 Das Plugin-Framework

Übersicht

Das Plugin Framework orientiert sich sehr an der Art, wie Eclipse mit seinen Plugins umgeht. Es setzt auf dem Service Framework auf, indem es definiert, wie neue Services, später als zur Kompilierungszeit, in das System integriert werden können. Es verwendet dazu das *Plugin* Architekturmuster.

Das Plugin Framework erweitert gezielt die Klassen und Schnittstellen des Service Frameworks, um seine Aufgabe zu erfüllen, die daraus besteht Services, Extensions und Jobs zur Startzeit von Matrix4.plot aus einer Jar⁸-Datei nachzuladen und in das System zu integrieren. Besonders ist hier zu beachten, dass die Jar-Dateien nicht Teil des Java Klassenpfades sind und so nicht automatisch von der Java VM⁹ geladen werden können.

Ein Plugin besteht aus einer einzelnen Jar-Datei, in der alle Klassen des Plugins enthalten sind. Zusätzlich befindet sich in dieser Jar-Datei die Datei *dotplotplugin.xml*, in der die Konfiguration des Plugins gespeichert ist.

Alle Jar-Dateien, die ein Plugin enthalten, werden im Plugin-Verzeichnis des Systems gespeichert und automatisch als solche erkannt. Das Plugin-Verzeichnis ist im PluginContext definiert, der die Klasse FrameworkContext für das Plugin Framework erweitert.

Verwaltet werden die Plugins ebenfalls über den PluginContext, in dem alle Plugins in der PluginRegistry registriert sind. Ein Plugin wird intern durch die Klasse Plugin repräsentiert. Die Klasse Plugin kümmert sich um die Services, Extensions und Jobs, die durch das Plugin in das System integriert werden und löst Versionskonflikte zwischen Plugins auf. Dadurch können mehrere Versionen eines Plugins im Plugin-Verzeichnis liegen, aber nur die neueste Version wird geladen.

Zusätzlich zu diesen Verwaltungsaufgaben stellt das Plugin Framework auch einige Services bereit, die sich um das Laden, Initialisieren und Integrieren der Plugins kümmern.

Der PluginLoadingService übernimmt das Laden der Klassen eines Plugins und erzeugt dabei die Plugin – Objekte für ihre Verwaltung. Insbesondere ist dieser Service für das Auflösen von Abhängigkeiten der Plugins untereinander zuständig. Plugins können Extensions für Services enthalten, die durch ein anderes Plugin in das System integriert werden. Dadurch ist es notwendig, dass der Service vor der Extension geladen wird.

Diese Abhängigkeiten werden in der *dotplotplugin.xml* Datei des Plugins festgehalten, und der PluginLoadingService löst die Abhängigkeiten auf, in dem er zunächst die *dotplotplugin.xml* Dateien der Plugins ausliest und anhand

8 Java Bytecode-Dateien werden in Jar-Dateien gebündelt. Technisch gesehen handelt es sich um normale Zip-Dateien.

9 Java Virtual Machine, die Java Laufzeitumgebung in der Java – Bytecode ausgeführt wird.

dieser die Reihenfolge festlegt, in der die Plugins geladen werden.

Sind viele Plugins im System vorhanden, kann es passieren, dass es zur Namenskollision zwischen zwei Klassen kommt. Um dem vorzubeugen, wird jedem Plugin ein eigener `ClassLoader` zugewiesen, der die Klassen des Plugins lädt. `ClassLoader` lassen sich zu Ketten mit einander verbinden, in der jeder `ClassLoader`, der eine Klasse nicht finden kann, den nächsten `ClassLoader` in der Kette fragen kann, ob dieser nicht die Klasse findet.

Die `ClassLoader` einzelnen Plugins werden vom `PluginLoadingService` anhand der Pluginabhängigkeiten vergeben, so dass die `ClassLoader` entsprechend der Abhängigkeit ihrer Plugins zusammenhängen. Auf diese Weise entsteht eine Baumstruktur mit dem `ClassLoader` des Systems als Wurzel und dem `ClassLoader` der meisten Abhängigkeiten als Blättern.

Der Raum, in dem es hier zu Namenskollisionen kommen kann, erstreckt sich nur entlang einzelner Äste. Kollisionen zwischen zwei Ästen sind nicht möglich.

Nach dem Laden erstellt der `PluginLoadingService` eine Plugin-Liste, die vom `PluginIntegrationService` weiterverarbeitet wird. Dieser Service integriert die Plugins in den `PluginContext`. Nach der Integration werden alle integrierten Services durch den `InitializerService` initialisiert. Danach ist das System bereit seine Arbeit aufzunehmen.

Durch die Trennung des Ladeprozesses in drei Einzelschritte, ist es leichter den Prozess zu einem späteren Zeitpunkt an neue Anforderungen anzupassen. Zum Beispiel wenn ein neuer Zwischenschritt eingeführt wird.

Der `InitializerService` besitzt als einziger der drei Services zwei Hotspots, den *Startup* und *Shutdown* Hotspot. An diesen Hotspots können Jobs registriert werden, die bei Systemstart bzw. bei Herunterfahren des System ausgeführt werden sollen. Das ermöglicht es den Systementwicklern auf beide Ereignisse reagieren zu können.

Jobs, die durch Plugins neu in das System kommen, werden im `PluginContext` mit Hilfe einer `JobRegistry` verwaltet.

Konfiguration von Plugins

Ein Plugin besteht aus Services, Erweiterungen und Jobs. Alle diese Klassen befinden sich in der Jar-Datei des Plugins und müssen dem System bekannt gemacht werden. Dies geschieht über die Datei *dotplotplugin.xml*, die sich ebenfalls in der Jar-Datei des Plugins befindet. Am Namen kann das System die Datei erkennen und die Konfiguration des Plugins laden.

Die Datei *dotplotplugin.xml* ist eine XML Datei die folgender DTD¹⁰ entspricht:

```
<!ELEMENT Dotplotplugin (Dependency*, Service*, (Job|Batchjob)*)>
<!ATTLIST Dotplotplugin id          CDATA #REQUIRED
                        name         CDATA #REQUIRED
                        version      CDATA #REQUIRED
                        provider     CDATA #IMPLIED
                        info         CDATA #IMPLIED>

<!ELEMENT Dependency EMPTY>
<!ATTLIST Dependency plugin CDATA #REQUIRED
                        version CDATA #REQUIRED>

<!ELEMENT Service (Extention*)>
<!ATTLIST Service id      CDATA #REQUIRED
                  class CDATA #IMPLIED>

<!ELEMENT Extention (Parameter*)>
<!ATTLIST Extention hotspot CDATA #REQUIRED
                  class  CDATA #REQUIRED
                  factory CDATA #IMPLIED>

<!ELEMENT Parameter EMPTY>
<!ATTLIST Parameter name  CDATA #REQUIRED
                  value CDATA #REQUIRED>

<!ELEMENT Job EMPTY>
<!ATTLIST Job id      CDATA #REQUIRED
              class CDATA #REQUIRED>

<!ELEMENT Batchjob (Task+)>
<!ATTLIST Batchjob id CDATA #REQUIRED>

<!ELEMENT Task EMPTY>
<!ATTLIST Task serviceid CDATA #REQUIRED>
```

Dotplotplugin

Hierbei handelt es sich um das Wurzelement des XML-Dokuments. Mit ihm werden die wichtigsten Metadaten des Plugins gesetzt. Die *id* identifiziert das Plugin systemweit eindeutig und über *name* wird ein vom Anwender leichter verständlicher Name für das Plugin vergeben. Die *id* sollte folgender Konvention folgen: [Domain des Providers].[Name des Plugins]

Mit *version* erhält das Plugin eine Versionsnummer, mit der die

¹⁰ Data Type Definiton: Bestimmt die korrekte Struktur von XML Dokumenten [W3CX]

Versionsabhängigkeiten der Plugins untereinander aufgelöst werden können.

Mit `provider` und `info` können zusätzliche Informationen über das Plugin angegeben werden. Der `provider` sollte eine Angabe über den Hersteller des Plugins enthalten, und `info` sollte aus einer Zusammenfassung der Funktionalität des Plugins bestehen.

Beispiel:

```
<Dotplotplugin
  id="org.dotplot.core.Standard"
  name="Standard"
  version="1.0"
  provider="FH Giessen-Friedberg (www.fh-giessen.de)"
  info="Standard extentions of Matrix4.plot.">
...
</Dotplotplugin>
```

Dependency

Mit diesem Element werden Abhängigkeiten von anderen Plugins angegeben. In der Regel handelt es sich hierbei um Plugins die Services bereitstellen, für die im aktuellen Plugin Erweiterungen angegeben sind.

Mit `plugin` wird das Plugin benannt, das unbedingt im System vorhanden sein muss, und mit `version` wird die minimal vorausgesetzte Version dieses Plugins angegeben.

Es können beliebig viele Abhängigkeiten angegeben werden, die alle erfüllt sein müssen bevor das Plugin im System integriert wird.

Beispiel:

```
<Dependency plugin="org.dotplot.core.Standard" version="1.0" />
```

Service

Mit diesem Element können neue Services und neue Erweiterungen in das System integriert werden. Angegeben werden muss eine `id`, die den Service identifiziert. Ist der Service im System nicht bekannt, kann mit dem Attribut `class` die Klasse des Service angegeben werden. In dem Fall wird ein Objekt der Klasse erzeugt und im System unter der angegebenen `id` registriert. Ist bereits ein Service unter der angegebenen `id` registriert, wird das `class` Attribut ignoriert. Der angegebene Service gilt für alle Extension Kindelemente.

Beispiel:

```
<Service id      ="org.dotplot.standard.FMatrix"
         class="org.dotplot.fmatrix.FMatrixService">
</Service>
```

oder

```
<Service id ="org.dotplot.standard.FMatrix">
</Service>
```

Extention

Mit diesem Element werden Erweiterungen in den als Elternelement angegebenen Service registriert. Hierzu muss mit `hotspot` der Hotspot für die Erweiterung angegeben werden und mit `class` die Klasse der Erweiterung. Diese Klasse muss zu dem Typ passen, den der Hotspot als Erweiterung erwartet. Zusätzlich kann noch eine Fabrik angegeben werden mit der die Erweiterung erzeugt werden soll. Wird sie ausgelassen wird die Erweiterung direkt erzeugt mit `newInstance()`.

Die in `factory` angegebene Klasse muss die Schnittstelle `org.dotplot.plugins.IExtentionFactory` implementieren.

Beispiel:

```
<Service id="org.dotplot.standard.Filter">
  <Extention hotspot="org.dotplot.standard.Filter.newFilter"
    class="org.dotplot.tokenizer.filter.LineFilter">
  </Extention>
</Service>
```

oder

```
<Service id="org.dotplot.standard.Filter">
  <Extention hotspot="org.dotplot.standard.Filter.newFilter"
    class="org.dotplot.tokenizer.filter.LineFilter"
    factory="org.dotplot.tokenizer.filter.FilterFactory">
  </Extention>
</Service>
```

Parameter

Mit diesem Element können zusätzliche Parameter an die Erweiterung übergeben werden. Sie sind über das `Extention` Objekt der Erweiterung zugänglich. Auf diese Weise kann die Erweiterung flexibler gestaltet werden und spezifischere Angaben über die Verwendung der Erweiterung gemacht werden.

Das Attribut `name` benennt den Parameter und mit dem Attribut `value` wird der Wert zugewiesen. Jeder Name kann nur einmal als Parameter angegeben werden.

Beispiel:

```
<Service id="org.dotplot.standard.Filter">
  <Extention hotspot="org.dotplot.standard.Filter.newFilter"
    class="org.dotplot.tokenizer.filter.LineFilter">
    <Parameter name="name" value="LineFilter"/>
    <Parameter name="ui" value="org.dotplot.LineFilterUI"/>
  </Extention>
</Service>
```

Job

Mit diesem Element werden neue Jobs im System registriert. Hierzu muss mit `id` ein Schlüssel angegeben werden unter dem der Job registriert wird und mit `class` wird die Klasse des Jobs angegeben. Beim Laden wird automatisch ein Objekt der Klasse, bei der es sich um eine Implementierung der Schnittstelle

org.dotplot.core.services.IJob handeln muss, angelegt und unter der Id registriert.

Beispiel:

```
<Job id="org.dotplot.jobs.TestJob" class="org.dotplot.TestJob" />
```

Batchjob

Ein Batchjob ist eine besondere Art von Job, der sich um das Abarbeiten von *Serviceketten* kümmert. Ihm wird eine Liste von Service Ids in Form des Elements Task übergeben, die beim Aufruf des Jobs in der übergebenen Reihenfolge abgearbeitet werden. Es können beliebig viele Tasks angegeben werden.

Mit dem Attribut id wird der Schlüssel angegeben unter dem der Batchjob registriert wird.

Beispiel:

```
<Batchjob id="org.dotplot.TestBatchjob"></Batchjob>
```

Task

Mit diesem Element und seinem Attribut serviceid werden dem Batchjob die Service Ids übergeben, die er abarbeiten soll. Hier ist die Reihenfolge der Angabe gleich der Reihenfolge der Abarbeitung.

Beispiel:

```
<Batchjob id="org.dotplot.TestBatchjob">
  <Task serviceid="org.dotplot.standard.Tokenizer" />
  <Task serviceid="org.dotplot.standard.Filter" />
  <Task serviceid="org.dotplot.standard.FMatrix" />
</Batchjob>
```

Muster: *Plugin*

Quelle: [PEAA]

Zweck

Verknüpft Klassen während der Konfiguration statt beim kompilieren.

Struktur

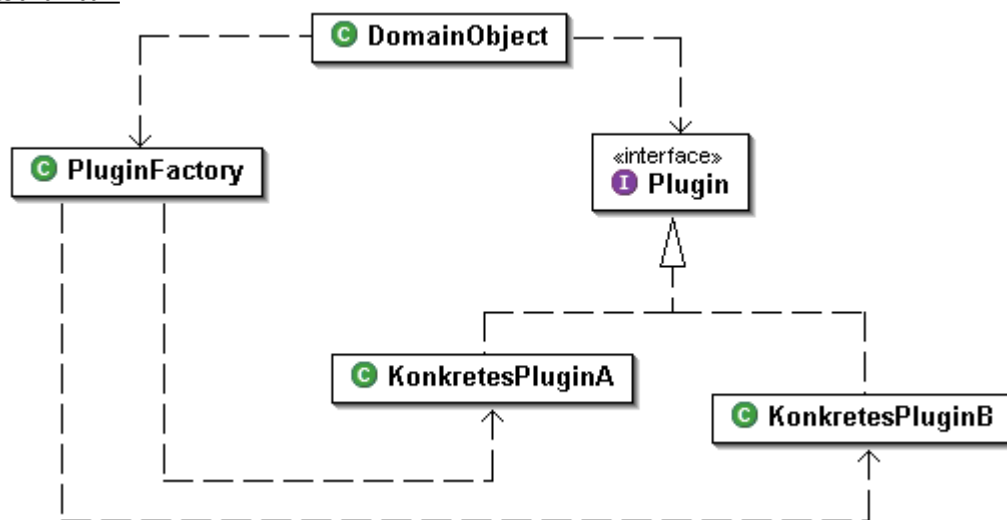


Diagramm 17 - Struktur des Musters Plugin

Teilnehmer

- **DomainObject**
 - fordert an der PluginFactory ein Plugin an.
- **Plugin**
 - stellt eine Schnittstelle bereit die von DomainObjekten benutzt werden kann.
- **PluginFactory**
 - bestimmt anhand der Konfiguration zur Laufzeit welches KonkretesPlugin Objekt zurückgegeben wird.
 - lädt gegebenenfalls neue Klassen in die Laufzeitumgebung.
- **KonkretesPlugin**
 - implementiert die Schnittstelle um von DomainObjekten benutzt zu werden.

Plugin in der Anwendung

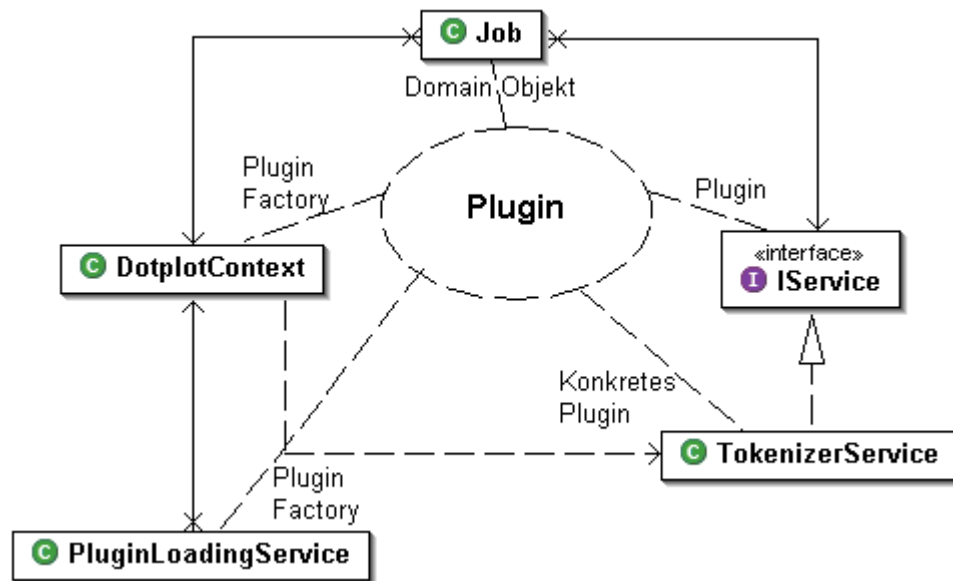


Diagramm 18 - Plugin in der Anwendung

Ein Job möchte einen Service, genauer den `TokenizerService` ausführen. Zu diesem Zweck benutzt er den `DotplotContext`, dem die Konfiguration für den `TokenizerService` bekannt ist.

Der `PluginLoadingService` lädt bei Systemstart die Konfiguration des `TokenizerService` aus einer Datei und teilt sie dem `DotplotContext` mit, die `Plugin Factory` ist also auf zwei Klassen verteilt.

7.4 Das Kernsystem

Aufbauend auf dem Plugin Framework ist das Kernsystem der Grundstein aller Dotplotanwendungen des Systems. Es versorgt das System sowohl mit einer an Dotplot angepassten Implementation des Plugin Frameworks, als auch mit dem ersten im System integrierten Plugin: *Core*.

Das *Core* Plugin unterscheidet sich von anderen Plugins darin, dass es fest im Quellcode implementiert ist und automatisch beim Start des Systems geladen wird. Es ist in der Klasse `org.dotplot.core.system.CoreSystem` implementiert und besteht aus den drei bereits vorgestellten Services: `PluginLoadingService`, `PluginIntegrationService` und `InitializerService` des Plugin Frameworks. Außerdem integriert das *Core* Plugin drei Jobs in das System, den `PluginLoadingJob`, den `StartupJob` und den `ShutdownJob`.

Der `PluginLoadingJob` bestimmt wie die Plugins in das System integriert werden, indem er eine *Servicekette* aus den drei Services ausführt. Der `StartupJob` holt sich die an den *Startup* Hotspot des `InitializerService` eingefügten Jobs und führt sie aus. Der `ShutdownJob` macht dasselbe mit den Jobs des *Shutdown* Hotspots.

Das *Core* Plugin gehört eigentlich nicht zur Dotplot Domäne, da es keinerlei für das Dotplot System angepassten Code besitzt. Seine Aufgabe ist es das System für die Aufnahme von Plugins bereit zu machen und diese Aufgabe tatsächlich durchzuführen. Erst mit dem Laden der Dotplot spezifischen Plugins wechselt das Kernsystem in die Domäne der Dotplots.

Die Integration des *Core* Plugins in das System wird von der Klasse `org.dotplot.core.ContextFactory` übernommen. Die Aufgabe der `ContextFactory` ist es den `DotplotContext` zu erstellen, der die Klasse `PluginContext` des Plugin Frameworks erweitert.

Die `ContextFactory` erzeugt hierzu ein Objekt der Klasse `DotplotContext` und integriert das *Core* Plugin in den Kontext, danach werden nacheinander der `PluginLoadingJob` und der `StartupJob` ausgeführt. Nach Ablauf dieses Prozesses ist der `DotplotContext` bereit um als Framework Kontext für das Dotplot System zu dienen.

Während der Laufzeit des Systems gibt es nur ein einziges `DotplotContext` Objekt, das von der `ContextFactory` verwaltet wird. Er ist mit der statischen Methode `getContext()` zugänglich. Tatsächlich wird der `DotplotContext` erst beim allerersten Aufruf dieser Methode erzeugt.

Der `DotplotContext` verwaltet alle wichtigen Parameter des Systems, wie Services, Plugins, Jobs. Zusätzlich zu diesen grundlegenden Diensten, die der Context von seinen Oberklassen erbt, gibt es eine `ConfigurationRegistry`, eine `TypeRegistry` und eine `TypeBindingRegistry`.

In der `ConfigurationRegistry` werden `Configuration` Objekte gespeichert, die Services dort registrieren können. Diese Konfigurationen werden

beim Herunterfahren des Systems in der Eclipse Konfiguration gespeichert und automatisch beim nächsten Start wieder hergestellt.

Die `TypeRegistry` verwaltet alle dem System bekannten Quelltypen von `Dotplots` in Form von `ISourceType` Objekten. Diese Quelltypen werden in der `TypeBindingRegistry` an die Endung von Dateinamen gebunden. Diese beiden Registries sind die Basis auf dem die Typisierung von `Dotplots` beruht.

Beispiel:

Die Klasse `TextType` steht für alle Arten von Text und ist in der `TypRegistry` mit dem Schlüssel `org.dotplot.types.Text` registriert. In der `TypeBindingRegistry` ist die Endung `.txt` an den Typ `org.dotplot.types.Text` gebunden.

Die Klasse `org.dotplot.core.DotplotService` ist ebenfalls Teil des Kerns und wird als Oberklasse aller im `Dotplot` System enthaltenen Services benutzt. Die einzigen Ausnahmen von dieser Regel sind die Services des *Core* Plugins. Alle zukünftigen Serviceentwickler müssen ihre Services von der Klasse `DotplotService` ableiten, um bestimmte Dienste des Systems nutzen zu können.

Zum einen stellt der `DotplotService` eine Schnittstelle bereit, über die das System automatisch ein `Configuration` Objekt in der `ConfigurationRegistry` des `DotplotContext` registriert. Auf diese Weise kann der `DotplotService` Konfigurationsinformationen in der `Konfigurationsregistry` des Systems ablegen, die dann mit GUI-Elementen manipuliert werden können.

Zum anderen implementiert der `DotplotService` das *Interceptor* Architekturmuster, um eine nachträgliche Erweiterbarkeit zu erreichen. Interceptoren stellen Punkte im Framework dar, an denen das Framework selbst erweitert werden kann. Hierzu fangen Interceptoren vorherbestimmte Ereignisse des Frameworks ab und führen eigenen Code aus, bevor das Framework mit der eigenen Arbeit fortfährt.

Der `DotplotService` kennt zwei solcher Ereignisse an die Interceptoren mit Hilfe von Hotspots eingefügt werden können. Die beiden Hotspots, die sich an den beiden Ereignissen orientieren sind der *Before* und *After* Hotspot. Interceptoren die am *Before* Hotspot angemeldet sind, führt der `DotplotContext` vor seiner eigentlichen Task aus und die Interceptoren des *After* Hotspots nach dem seine Task beendet ist.

Auf diese Weise kann die Funktionalität eines Service erweitert werden, indem eine Manipulation des Arbeitskontexts und des Ergebniskontexts möglich wird.

Serviceentwickler können durch die Verwendung von Hotspots eigene Ereignisse definieren auf die mit den Interceptoren reagiert werden kann.

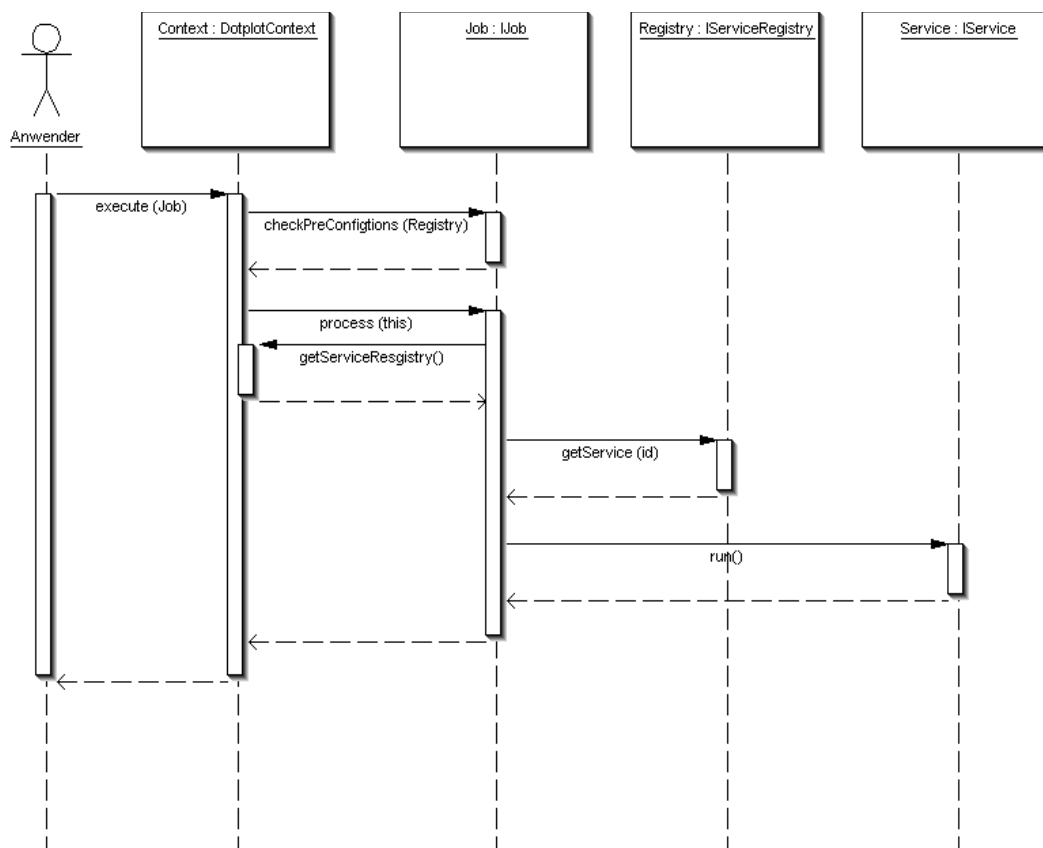


Diagramm 19 - Aufruf eines Jobs

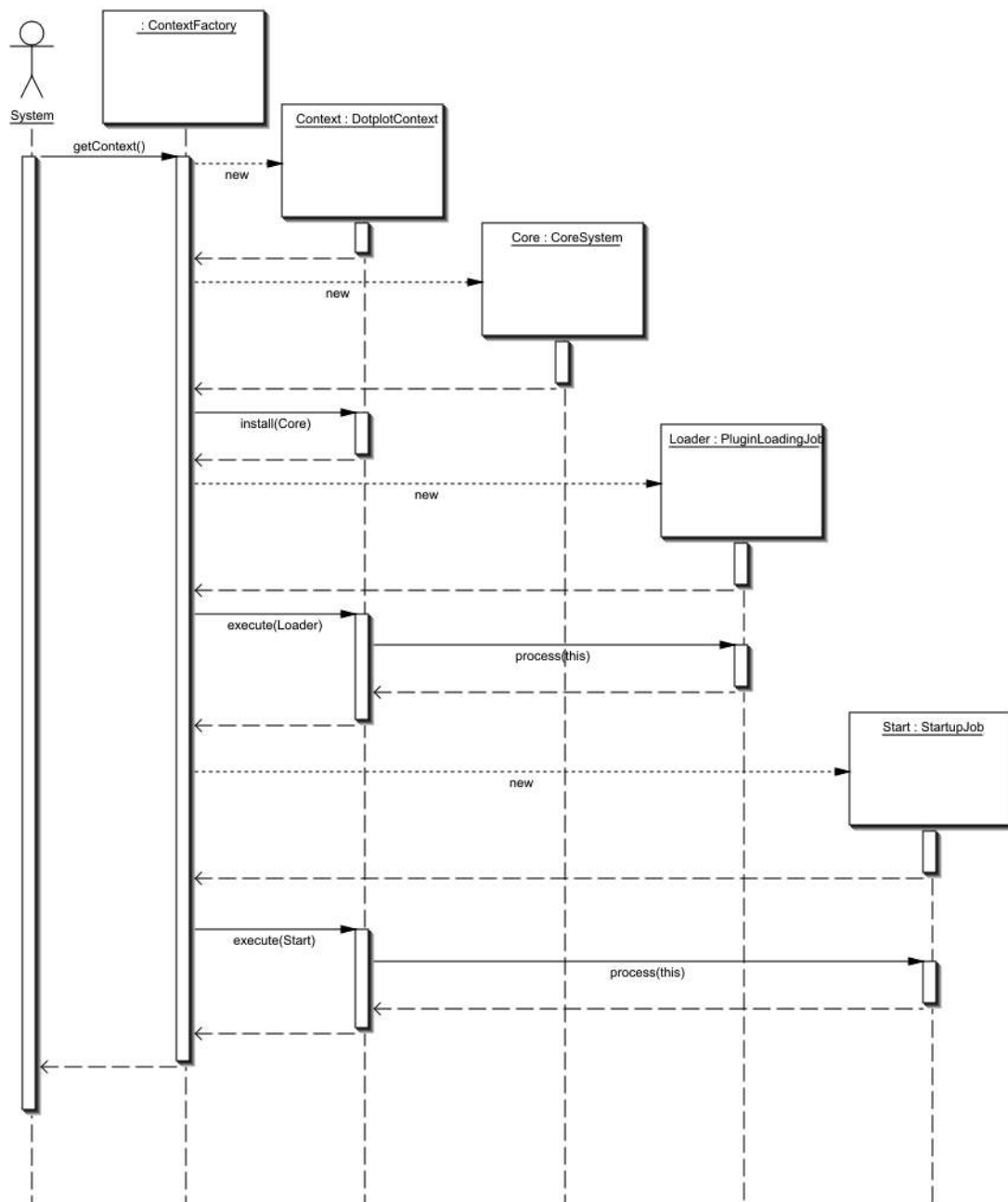


Diagramm 20 - Erzeugen des DotplotContext

Muster: *Interceptor*

Quelle: [PoSaII]

Zweck

Erlaubt es Dienste transparent zu einem Framework hinzuzufügen, die beim Auftreten bestimmter Ereignisse automatisch ausgeführt werden.

Struktur

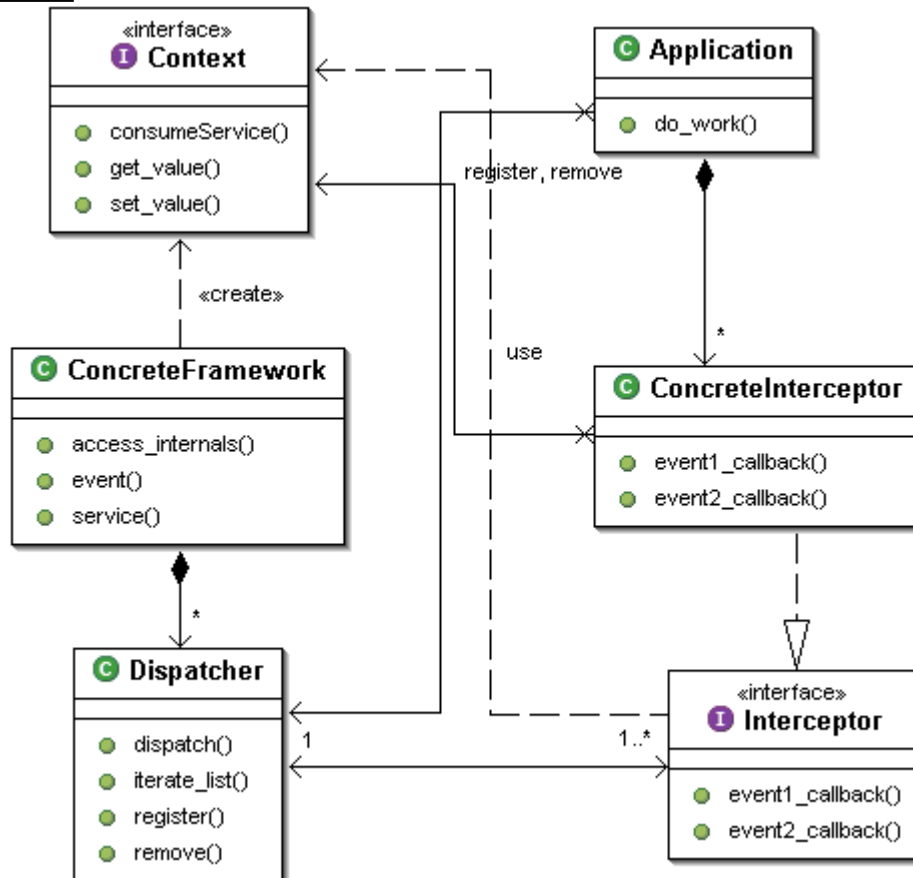


Diagramm 21 - Struktur von Interceptor

Teilnehmer

- **Context**
 - erlaubt es Diensten Informationen über das konkrete Framework zu erhalten.
 - Erlaubt es Diensten das Verhalten des konkreten Framework zu beeinflussen.
- **ConcreteFramework**
 - definiert Dienste
 - integriert Dispatcher um Anwendungen das abfangen von Events zu ermöglichen.
 - delegiert Events an die jeweiligen Dispatcher.

- **Dispatcher**
 - erlaubt es Applikationen konkrete Interceptoren zu registrieren oder zu entfernen.
 - ruft die Callback-Methoden von registrierten Interceptoren auf.
- **Application**
 - läuft über dem konkreten Framework.
 - implementiert konkrete Interceptoren und registriert diese an den Dispatchern.
- **Interceptor**
 - Definiert eine Schnittstelle für einen Dienst.
- **ConcreteInterceptor**
 - implementiert einen bestimmten Dienst.
 - benutzt ein Context – Objekt um das konkrete Framework zu kontrollieren.

Anwendung von *Interceptor*

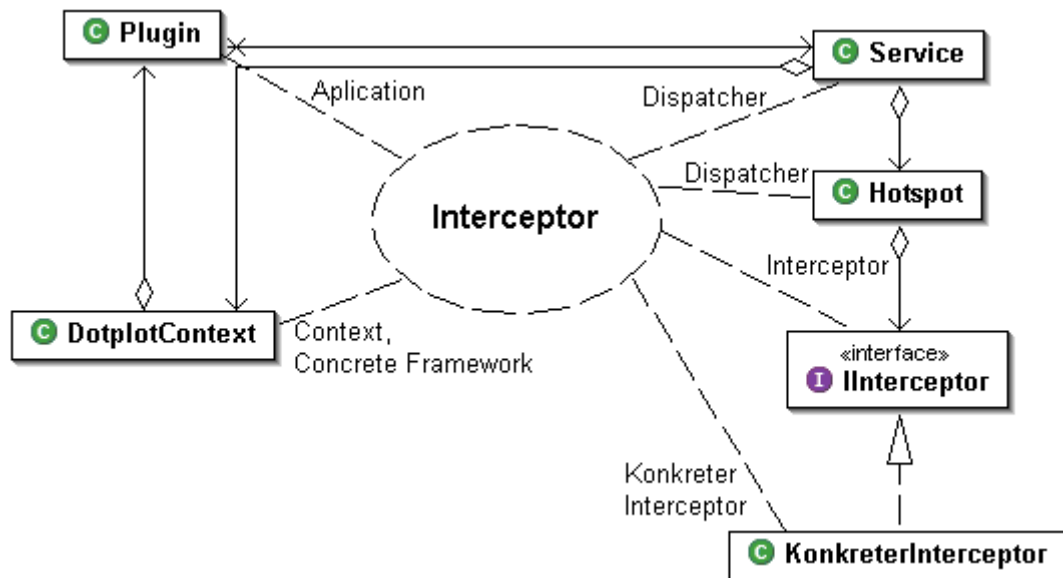


Diagramm 22 - Anwendung von *Interceptor*

Interceptor wird angewendet um die Funktionalität eines Service zu erweitern. Der Service arbeitet dabei mit einem Hotspot zusammen, wobei der Service die Events auslöst, auf die die Interceptoren reagieren können, die vom Hotspot verwaltet werden.

Der DotplotContext bildet hierbei sowohl den Context in dem ein Interceptor ausgeführt wird, als auch das Concrete Framework das vom Interceptor manipuliert werden kann.

Das Plugin, zu dem der Service und der Interceptor gehört, meldet die konkreten Interceptoren beim Service an.

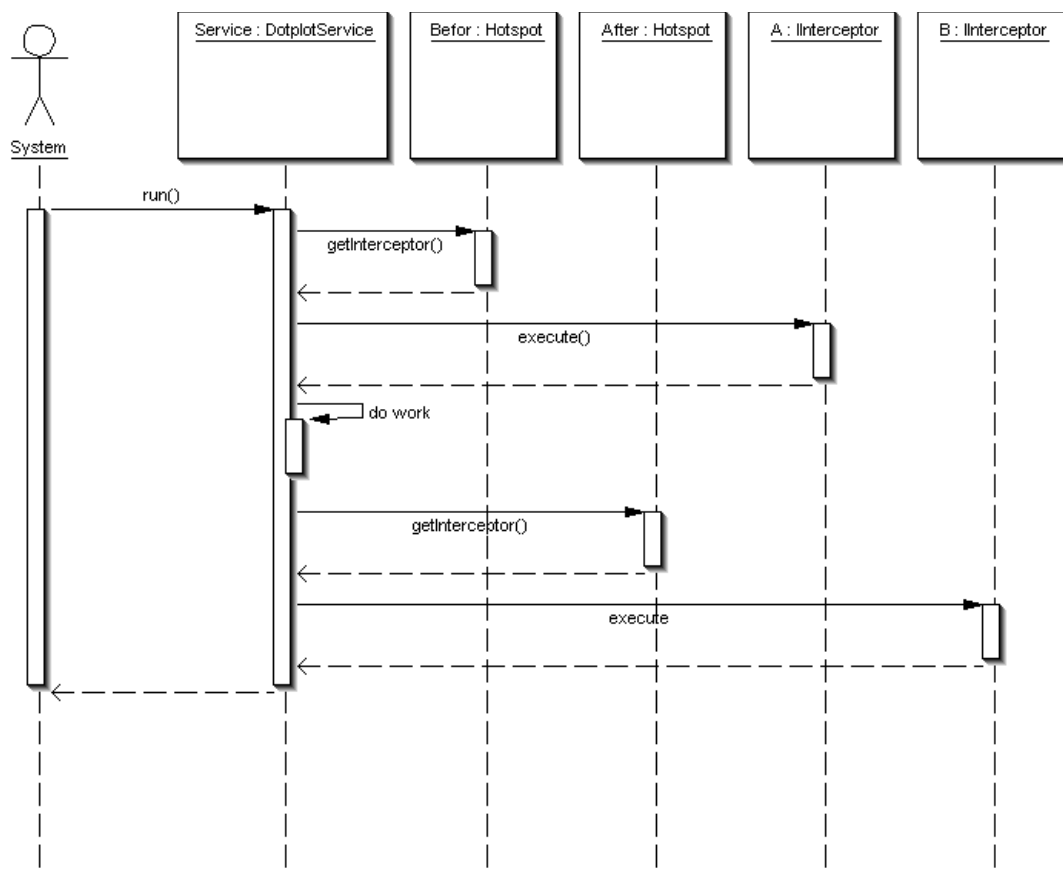


Diagramm 23 - Aufruf der Interceptoren

Hier kann man sehen, wie die Interceptoren bei der Ausführung eines Service aufgerufen werden.

Zunächst werden die Interceptoren ausgeführt, die sich am *Before* Hotspot angemeldet haben. Danach erledigt der Service seine eigentliche Aufgabe, mit dem erzeugen seiner Task und dem Aufruf des TaskProcessors. Nachdem die Task bearbeitet ist, werden die Interceptoren des *After* Hotspots ausgeführt.

7.5 Anpassungen an die Neue Architektur

Die von der alten Architektur bereits bekannten Teilschritte des *Pipes & Filters* Musters sind: der Tokenizer, die F-Matrix und das Q-Image. Diese drei Teilschritte sind als Pakete auch in der Codestruktur von Matrix4.plot zu erkennen. Zusätzlich dazu kommen noch die GUI-Elemente, die ein viertes Paket belegen und das fünfte Paket für das Grid-Computing. Diese fünf Pakete sollten sich in der neuen Paketstruktur des Systems wiederfinden lassen und sind darüberhinaus hervorragende Kandidaten für Services.

Tatsächlich wurden es ein paar mehr Services. Der Tokenizer wurde noch einmal in drei Einzelservices geteilt, den Konverter, den eigentlichen Tokenizer und den Filter. F-Matrix, Q-Image und GUI bekamen jeweils einen Service und das Grid bekam keinen Service. Der Grund dafür war, dass es sich beim Grid um einen grundlegenden Dienst des Frameworks handelt, der nicht durch einen Pluginmechanismus in das System eingeführt werden darf.

Die entstandenen Services im einzelnen

Der Konverter

Übersicht	
ID	org.dotplot.standard.Converter
Klasse	org.dotplot.tokenizer.converter.ConverterService
Arbeitskontext	org.dotplot.tokenizer.converter.SourceListContext
Resultatskontext	org.dotplot.tokenizer.converter.SourceListContext
Konfigurations ID	org.dotplot.converter.Cofiguration
Hotspots	
ID	newType
Klasse	org.dotplot.core.ISourceType
Parameter	name – der Name, unter dem der neue Typ im System registriert wird. suffix – die Dateiendung an die der Typ gebunden wird.
Beschreibung	Mit diesem Hotspots werden neue Typen eingeführt und gebunden. Das Binden von mehreren Endungen an einen Typ erfolgt durch ein weiteres Einfügen der Extention in die XML Datei mit den gleichen Parametern aber mit dem neuen Suffix.
ID	newConverter
Klasse	org.dotplot.tokenizer.converter.IConverter
Parameter	keine
Beschreibung	Mit diesem Hotspot können neue Konverter in das System eingefügt werden.
Beschreibung	
<p>Der Service des Konverters kümmert sich darum Quelldaten in ein anderes Datenformat zu konvertieren, um auf diesem Datenformat zusätzliche Tokenizer oder Filter anwenden zu können.</p> <p>Zusätzlich können über diesen Service neue Typen eingefügt und an Dateiendungen gebunden werden.</p>	

Der Tokenizer

Übersicht	
ID	org.dotplot.standard.Tokenizer
Klasse	org.dotplot.tokenizer.service.TokenizerService
Arbeitskontext	org.dotplot.tokenizer.converter.SourceListContext
Resultatskontext	org.dotplot.tokenizer.service.TokenStreamContext
Konfigurations ID	org.dotplot.tokenizer.Configuration
Hotspots	
ID	newTokenizer
Klasse	org.dotplot.tokenizer.service.ITokenizer
Parameter	id – um den Tokenizer im System eindeutig zu identifizieren. name – ein für den Anwender verständlicher Bezeichner.
Beschreibung	Mit diesem Hotspots werden neue Tokenizer in das System eingefügt.
Beschreibung	
Der Service des Tokenizers kümmert sich darum die Eingabedaten für die Weiterverarbeitung vorzubereiten, indem der richtige Tokenizer zugewiesen und der Tokenstrom bereit gemacht wird.	

Der Filter

<i>Übersicht</i>	
ID	org.dotplot.standard.Filter
Klasse	org.dotplot.tokenizer.filter.FilterService
Arbeitskontext	org.dotplot.tokenizer.service.TokenStreamContext
Resultatskontext	org.dotplot.tokenizer.service.TokenStreamContext
Konfigurations ID	org.dotplot.filter.Configuration
<i>Hotspots</i>	
ID	newFilter
Klasse	org.dotplot.tokenizer.filter.ITokenFilter
Parameter	name – um den Filter im System zu identifizieren. ui – eine Klasse, deren Objekte als GUI für die Einstellungen des Filters dient. Diese Klasse muss das Interface org.dotplot.tokenizer.filter.ui.IFilterUI implementieren.
Beschreibung	Mit diesem Hotspots werden neue Tokenizer in das System eingefügt.
<i>Beschreibung</i>	
Der Service des Filters kümmert sich um das Aufstellen einer Filterkette, um den Tokenstrom des Tokenizers zu filtern.	

Die F-Matrix

<i>Übersicht</i>	
ID	org.dotplot.standard.FMatrix
Klasse	org.dotplot.fmatrix.FMatrixService
Arbeitskontext	org.dotplot.tokenizer.service.TokenStreamContext
Resultatskontext	org.dotplot.fmatrix.FMatrixContext
Konfigurations ID	org.dotplot.fmatrix.Configuration
<i>Hotspots</i>	
keine	
<i>Beschreibung</i>	
Der Service der F-Matrix kümmert sich um die Generierung der F-Matrix aus dem Tokenstrom.	

Das Q-Image

Übersicht	
ID	org.dotplot.standard.QImage
Klasse	org.dotplot.image.QImageService
Arbeitskontext	org.dotplot.fmatrix.FMatrixContext
Resultatskontext	org.dotplot.image.QImageContext
Konfigurations ID	org.dotplot.qimage.Configuration
Hotspots	
keine	
Beschreibung	
Der Service des Q-Images erstellt den Dotplot aus der F-Matrix.	

Die GUI

Übersicht	
ID	org.dotplot.standard.EclipseUI
Klasse	org.dotplot.eclipse.EclipseUIService
Arbeitskontext	org.dotplot.core.services.IContext
Resultatskontext	org.dotplot.core.services.IContext
Konfigurations ID	keine
Hotspots	
ID	View
Klasse	org.dotplot.ui.ConfigurationView
Parameter	id – um den ConfigurationView eindeutig zu identifizieren.
Beschreibung	Mit diesem Hotspot können neue Karteikarten in den Dotplot-Konfigurationsdialog eingefügt werden.
ID	Menu
Klasse	org.dotplot.core.services.Strukture
Parameter	id – um den Menü eindeutig zu identifizieren. name – der Name des Menüs in der Menüleiste. menu – die id des Eltern-Menüs.

Beschreibung	Mit diesem Hotspot lassen sich neue Menüs in die Menüleiste einfügen. Dazu benötigt jedes Menü eine id und einen Namen. Wird ein Eltern-Menü angegeben, entsteht in diesem Eltern-Menü durch das neue Menü ein Submenü.
ID	Entry
Klasse	<code>org.dotplot.core.services.Strukture</code>
Parameter	name – der Name des Eintrags in der Menüleiste. menu – die id des Eltern-Menüs. job – die Klasse des auszuführenden Jobs.
Beschreibung	Dieser Hotspot fügt Einträge in Menüs ein und sorgt dafür, dass bei der Auswahl des Menüeintrags der angegebene Job ausgeführt wird.
<i>Beschreibung</i>	
Dieser Service verwaltet die GUI von Matrix4.plot, die unabhängig von der Eclipse Platform funktioniert. Besonders kümmert er sich um neue Menüeinträge, mit denen die Verarbeitung von Jobs ausgelöst werden, und die Gestaltung des Dotplot-Konfigurationsdialogs.	

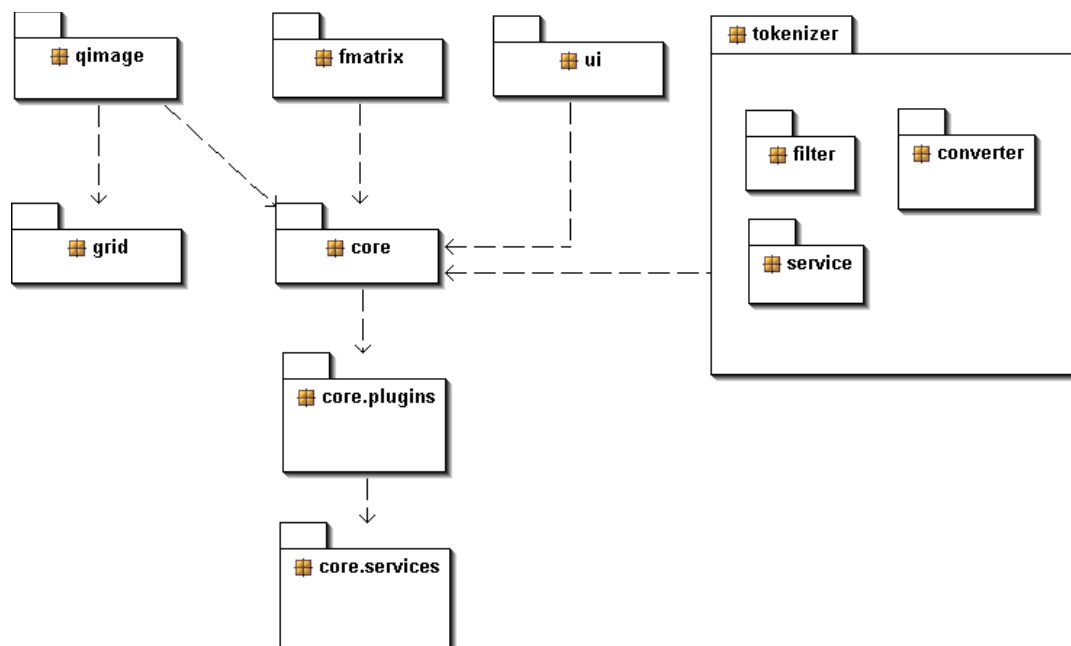


Diagramm 24 - Paketstruktur der neuen Architektur

Dieses Diagramm zeigt die Paketstruktur der neuen Architektur. Zu erkennen ist, wie das Core-Paket auf dem Plugin- und dem Service-Paket aufbaut, die das Plugin bzw. Service Framework enthalten.

Alle anderen Pakete bauen direkt auf dem Core-Paket auf wobei das Grid, ein wenig isoliert, nur vom Q-Image benutzt wird.

7.6 Portierung auf die Rich Client Plattform

Die Portierung von Matrix4.plot auf die Rich Client Plattform gestaltete sich mit Eclipse 3.1 erfreulich einfach. Das Plugin musste dazu um zwei Erweiterungen an den Punkten `org.eclipse.core.runtime.applications` und `org.eclipse.core.runtime.products` ergänzt werden.

Ein Product kümmert sich um die Grundeinstellungen der RCP bestehend aus der auszuführenden Application und Anpassungen an das Aussehen der Plattform, wie Fenster und Launcher Icons. Diese Einstellungen werden innerhalb einer gesonderten Datei mit der Endung „product“ gespeichert.

Eine Applikation stellt einen Einsprungpunkt in die Applikation dar, an dem der Programmablauf nach der Initialisierung beginnt. An diesen Erweiterungspunkt wird eine Klasse vom Typ `IPlatformRunnable` übergeben.[RCE]

DotplotApplication

Diese Klasse ist der Einsprungpunkt in die RCP Applikation Matrix4.plot. Nach ihrer Anmeldung an den Erweiterungspunkt `org.eclipse.core.runtime.applications` kann die Applikation als Startpunkt für die Plattform benutzt werden. Die Plattform wird dazu nach ihrer Initialisierung die `run()` Methode aufrufen. Die `run()` Methode macht nichts anderes als die Workbench zu starten und die Ereignisbehandlung der GUI in Gang zu setzen. Um die Workbench zu konfigurieren wird der Plattform beim Start ein `WorkbenchAdvisor` Objekt übergeben, in diesem Fall ein `DotplotAdvisor`. [RCE]

```
public class DotplotApplication implements IPlatformRunnable {

    public Object run(Object args) throws Exception {
        Display display = PlatformUI.createDisplay();
        int returnCode = PlatformUI.
            createAndRunWorkbench(display, new DotplotAdvisor());
        return (returnCode == PlatformUI.RETURN_RESTART)
            ? IPlatformRunnable.EXIT_RESTART
            : IPlatformRunnable.EXIT_OK;
    }
}
```

DotplotAdvisor

Ein `WorkbenchAdvisor` konfiguriert die Workbench und reagiert auf bestimmte Workbench Ereignisse. Diese Ereignisse sind Startup und Shutdown der Workbench, Ausnahmebehandlung der Ereignisschleife und die Initialisierung. [RCE]

Matrix4.plot benutzt den `DotplotAdvisor` um die Perspektive festzulegen, die nach dem Start automatisch angezeigt wird und sich zu initialisieren.

Die in der `initialize()` Methode vorgenommene Initialisierung von Matrix4.plot betrifft die `ContextFactory`, die mit den richtigen Parametern versorgt wird um den `DotplotContext` zu erzeugen. Im Wesentlichen werden

die Schemadatei, die das XML Schema¹¹ für die *dotplotplugin.xml* enthält, und das Plugin Verzeichnis in den Ressourcen des Plugins gesucht und der ContextFactory zugewiesen. Zum Abschluss wird über ContextFactory.getContext() der DotplotContext erzeugt.

Des weiteren wird die Methode preShutdown() benutzt bei Beendigung der Workbench den ShutdownJob auszuführen.

Über createWorkbenchWindowAdvisor() erzeugt der DotplotAdvisor den WorkbenchWindowAdvisor, der das Workbench Fenster konfigurieren soll.

```
public class DotplotAdvisor extends WorkbenchAdvisor {

    public String getInitialWindowPerspectiveId() {
        return "org.dotplot.plugin.perspective1";
    }

    public void initialize(IWorkbenchConfigurer configurer){
        //Location for DotplotPlugins
        Path pluginsPath = new Path("/plugins");
        //Location for Plugins Shemafile
        Path shemaPath = new Path("/ressources/dotplotschema.xsd")
        //Geting Platformindependent URL
        URL indepPluginsURL =
            DotplotPlugin.getDefault().find(pluginsPath);
        URL indepShemaURL =
            DotplotPlugin.getDefault().find(shemaPath);
        try {
            //Resolving Platformdependent URL
            URL depPluginsURL = Platform.resolve(indepPluginsURL);
            URL depShemaURL = Platform.resolve(indepShemaURL);

            ContextFactory.setPluginDirectory(depPluginsURL.getPath());
            ContextFactory.setShemaFile(depShemaURL.getPath());
            ContextFactory.getContext();
        }
        catch (IOException e) { }
    }

    public boolean preShutdown(){
        DotplotContext context = ContextFactory.getContext();
        try {
            context.executeJob(CoreSystem.JOB_SHUTDOWN_ID);
        }
        catch (UnknownIDException e) {}
        return true;
    }

    public WorkbenchWindowAdvisor createWorkbenchWindowAdvisor
        (IWorkbenchWindowConfigurer configurer){
        return new DotplotWindowAdvisor(configurer);
    }
}
```

11 Genau wie eine DTD bestimmt ein Schema den korrekten Aufbau einer XML-Datei. Ein Schema ist allerdings wesentlich flexibler als ein DTD, mit dem sich genauere Datentypen definieren lassen. [W3CS]

DotplotWindowAdvisor

Ein `WorkbenchWindowAdvisor` kümmert sich um die Konfiguration des Fensters der Anwendung. Auch hier gibt es Methoden, um auf Fensterereignisse wie `Open`, `Close` und `Restore` zu reagieren. Dem Konstruktor wird ein `WindowConfigurer` übergeben, über den das Aussehen des Anwendungsfensters und seiner `Shell` eingestellt werden kann. [RCE]

`Matrix4.plot` benutzt die Methode `createActionBarAdvisor()`, um das Hauptmenü der Anwendung einzustellen. Dazu wird ein `DotplotActionBarAdvisor` zurückgegeben.

```
public class DotplotWindowAdvisor extends WorkbenchWindowAdvisor {  
    public DotplotWindowAdvisor(  
        IWorkbenchWindowConfigurer configurer) {  
        super(configurer);  
    }  
  
    public ActionBarAdvisor createActionBarAdvisor  
        (IActionBarConfigurer configurer){  
        return new DotplotActionBarAdvisor(configurer);  
    }  
}
```

DotplotActionBarAdvisor

Ein `ActionBarAdvisor` konfiguriert die Menüleiste der Workbench. Mit der Methode `makeActions()` registriert der `DotplotActionBarAdvisor` von `JFace` bereitgestellte `Actions`. Diese `Actions` sind Standardmenüpunkte die auch von der Eclipse IDE verwendet werden. Für `Matrix4.plot` werden die *Quit*-Action zum Beenden der Workbench, die *About*-Action um den About-Dialog anzuzeigen und die *Preferences*-Action, um den Preferences-Dialog zu öffnen registriert. [RCE]

In `fillMenuBar()` wird zunächst der `MenuManager` für den Menüpunkt *File* erzeugt und am Hauptmenü registriert und danach die vorher erzeugten Standard-`Actions` in das File-Menü eingetragen.

Zum Schluss wird das Hauptmenü dem `UIService` übergeben, um die durch Dotplot Plugins ins System eingeführten Menüpunkte zu erzeugen.

```

public class DotplotActionBarAdvisor extends ActionBarAdvisor {

    public DotplotActionBarAdvisor(IActionBarConfigurer configurer) {
        super(configurer);
    }

    public void makeActions(IWorkbenchWindow window) {
        this.register(ActionFactory.QUIT.create(window));
        this.register(ActionFactory.ABOUT.create(window));
        this.register(ActionFactory.PREFERENCES.create(window));
    }

    public void fillMenuBar(IMenuManager menuBar) {

        MenuManager fileMenu =
            new MenuManager("&File", IWorkbenchActionConstants.M_FILE);

        menuBar.add(fileMenu);

        fileMenu.add(
            this.getAction(ActionFactory.PREFERENCES.getId()));
        fileMenu.add(this.getAction(ActionFactory.ABOUT.getId()));
        fileMenu.add(this.getAction(ActionFactory.QUIT.getId()));

        DotplotContext context = ContextFactory.getContext();
        try {
            EclipseUIService service =
                (EclipseUIService) context.getServiceRegistry().
                    get(EclipseConstants.ID_SERVICE_ECLIPSE_UI);
            service.fillMenuBar(menuBar);
        }
        catch (UnknownIDException e) {}
    }
}

```

8. Resümee

8.1 Möglichkeiten der neuen Architektur

Die neue Architektur baut auf Komponenten auf, die nicht aus der Dotplot Domäne kommen, so dass sich das Plugin- und das Service-Framework auch für andere Projekte verwenden lassen.

Durch die auf Plugins aufbauende Architektur ist Matrix4.plot bereit für seinen zukünftigen Einsatz in weiterführenden Projekten, die neue Funktionen in das Programm einführen.

Zukünftige Entwicklungsteams brauchen sich nur noch wenig mit dem Sourcecode des Programms zu befassen und können sich gezielter auf ihre Aufgaben konzentrieren.

Die Trennung der einzelnen Komponenten in Services ermöglicht es den Dotplot-Generierungsprozess gezielter zu manipulieren. So wird es möglich bei der Generierung direkt auf einer F-Matrix aufzusetzen, ohne die Plotquellen erneut einlesen zu müssen.

Neue Services können transparent in das System einfließen, ohne das bestehende Services beeinträchtigt werden, genauso verhält es sich mit den Interceptoren, die es ermöglichen das Verhalten eines Service nachträglich zu beeinflussen, ohne an dem Quellcode des Service etwas ändern zu müssen.

Durch den Umstand das sich Plugins gezielt aktivieren und deaktivieren lassen, kann der Anwender sehr direkt in das System, und seine Funktionen eingreifen.

Tasks und Taskprozessoren entkoppeln Services von ihrer Ausführung gestalten diese Ausführung in großem Maß flexibel.

Durch die Definition von Jobs und deren Integration in die Menüleiste können Plugin-Entwickler direkt neue Funktionen in die graphische Oberfläche einbauen. Dabei sind sie durch die offene Definition eines Jobs an keine Konventionen gebunden, die sie in der Implementierung der neuen Funktionen behindern.

Plotquellen und deren Typisierung ermöglicht die flexible Erweiterung des Funktionsumfangs des bestehenden Dotplotmechanismus. Neue Tokenizer und Filter lassen sich Transparent auf die neuen Plotquellen zuschneiden, ohne das andere Komponenten des Dotplotmechanismus beeinträchtigt werden.

Abschließend ist zu bemerken, dass man selbst in einem solch fortgeschrittenen Projekt wie Matrix4.plot durch die Verwendung von Entwicklungsmustern der Softwareentwurf und damit die Softwarearchitektur maßgeblich verbessert werden kann. Das gilt insbesondere in den Bereichen Flexibilität und Struktur.

8.2 Erfahrungen mit Testfirst und JUnit

Die Erfahrungen mit JUnit zeigen, dass es ein hervorragendes Mittel ist, um automatische Tests durchzuführen. Daher eignet sich JUnit für den Testfirstansatz sehr gut.

Allerdings zeigt die Praxis auch, dass ein gehöriges Maß an Disziplin dazugehört den Testfirstansatz in all seiner ganzen Konsequenz durchzuhalten. Der natürliche Arbeitsfluss muss für das Schreiben eines neuen Tests unterbrochen werden, wenn man unvorhergesehen eine neue Klasse braucht. In der Regel unterbricht man seine Programmierarbeit dazu jedoch nicht und neigt dazu die Tests im Anschluss an einen fertig gestellten Programmabschnitt zu schreiben. Dabei ergeben sich allerdings auch einige Vorteile. Zum einen kann man immer noch seinen Code testen, um Flüchtigkeitsfehler aufzudecken, was auch beim Auftreten von Seiteneffekten¹² auf andere Programmteile hilfreich ist. Bestehende Tests schlagen dann unvorhergesehen fehl. Zudem gibt es die Möglichkeit einen automatischen Fehlschlag in Tests einzubauen, um noch zu implementierende Programmteile zu markieren. Dadurch kann das Fortschreiten der Arbeit direkt an der JUnit Anzeige verfolgt werden.

Das führt auch zu einer Art „Mini Testfirst“ in dem neue Testklassen nicht komplett, sondern nur die Tests zu den Methoden, an denen gerade arbeitet wird, implementiert werden.

Abschließend ist noch anzumerken, dass der Testfirstansatz sehr von der Erfahrung des Testschreibers ab, sinnvolle Tests zu schreiben, nur solche Fehler können gefunden werden, auf die auch berücksichtigt wurden. So werden Tests auf falsche Eingaben von einem ungeübten Testschreiber schnell übersehen, oder Trivialtests die zwar korrekt aber eigentlich überflüssig sind, erschweren es die Übersicht über die Funktionsweise eines Tests zu behalten. Das kann die Fehlersuche erschweren.

8.3 Ausblick

Auch wenn Matrix4.plot nun für seine zukünftige Entwicklung gewappnet ist, so bleiben einige Dinge noch zu tun, auf die in dieser Diplomarbeit noch nicht eingegangen werden konnte.

Das Grid

Besonders hervorzuheben ist da das Grid-Computing, das im Moment, wie in der alten Version, nur für die Erstellung des Dotplots aus der F-Matrix verwendet wird. Die neue Architektur bietet die Möglichkeit das Grid umfassender einzusetzen. Die Schnittstellen, die durch das Interface `ITaskProcessor` bereitgestellt werden sind dafür prädestiniert.

`Tasks` lassen sich von Haus aus in `TaskParts` aufteilen, die unabhängig voneinander bearbeitet werden können. Es spricht nichts dagegen, die `TaskParts` über das Grid verteilt berechnen zu lassen.

¹² Ein Seiteneffekt beschreibt den Umstand, das die Änderung an einer Komponente Auswirkungen auf eine andere Komponente haben können. Seiteneffekte treten meist unvorhergesehen auf.

Die wesentlichen hierfür zu bewältigenden Aufgaben sind:

- Das Grid-Protokoll muss eine Verteilung von `TaskParts` erlauben.
- Grid-Clients müssen eingehende `TaskParts` verarbeiten können und eventuell benötigte Ressourcen vom Server nachladen und synchronisieren.
- Ein `TaskProcessor` für den Server muss erstellt werden, der die `TaskParts` seiner Task an die Grid-Clients verschickt, wenn die Tasks teilbar sind.
- Letztendlich braucht das verwendete Grid-Computing noch ein gutes Fehlerbehandlungsprotokoll.

Das GUI-System

Der nächste große Brocken der noch bewältigt werden muss, ist ein neues GUI-System. Ziel der neuen GUI muss es sein, die Flexibilität widerzuspiegeln, die durch den Pluginmechanismus in das neue System aufgenommen wurde.

Für die neue Architektur wurde das alte GUI System soweit angepasst, dass es einigermaßen auf die sich ändernden Komponenten reagieren kann. Es handelt sich hierbei allerdings nur um ein Provisorium und sollte in Zukunft noch überarbeitet werden.

Ziel muss es sein, das GUI System an den Services auszurichten. Jedem Service wird eine GUI Komponente zugewiesen, aus denen ein Taskprocessor, abhängig von den Services die er in einer *Servicekette* bearbeitet, einen Konfigurationsdialog zusammenbaut. Dieser Konfigurationsdialog sollte sich an einem von Wizards her bekannten Verhaltensmuster orientieren, das den Anwender durch die einzelnen Dialoge führt.

Im Moment orientiert sich das GUI System an dem Verhaltensmuster von Karteireitern und es kann schnell passieren, dass der Anwender bestimmte Einstellungen vergisst. Insbesondere können Einstellungen von einer Karteikarte nur durch Drücken auf den Apply-Knopf für eine andere Karteikarte bekannt gemacht werden, was schnell vergessen werden kann.

Diese Informationsweitergabe von einer GUI Komponente zur nächsten ist es, die das neue GUI Framework besser unterstützen soll, so dass die einzelnen Seiten des Wizards besser aufeinander aufbauen können, als das im Moment der Fall ist.

Das Hilfe-System

Aktuell gibt es keinerlei Hilfe für den Benutzer, er wird mit dem Programm alleine gelassen. Auch hierfür kann Eclipse Abhilfe schaffen.

Eclipse unterstützt mit mehreren Extension Points verschiedene Methoden der Onlinehilfe, einerseits die typische übersichtliche Buchstruktur, aber auch eine kontextsensitive Hilfe die direkt auf Mausklick die Hilfe zu bestimmten Situationen und Dialogen öffnen kann.[RCE]

Die Online Hilfe von Eclipse basiert auf HTML Dokumenten und unterstützt diese auch in verschiedenen Sprachversionen.

Automatische Onlineupdates

Hierbei handelt es sich um eine Funktion, die ebenfalls erst kürzlich mit Version 3.0 in die Eclipse Plattform eingeführt wurde. Mit deren Hilfe lassen sich die Eclipse-Features, welche Matrix4.plot in der RCP-Version verwendet aktualisieren. Hierbei kann eingestellt werden, ob das Update automatisch bei Systemstart erfolgen soll, oder manuell durch den Anwender initiiert wird.[RCE]

Um Onlineupdates zu realisieren, muss zunächst die Infrastruktur geschaffen werden, um die Updates online verfügbar zu machen. Die Anpassungen an Matrix4.plot lassen sich unabhängig von übrigen Code des Programms durchführen und befasst sich mit der Implementation einer entsprechenden UpdateAction die von Eclipse ausgeführt werden kann.

Diese UpdateAction lässt sich in die Menüleiste des RCP einbauen, um einen manuellen Start zu unterstützen. Um ein automatisches Update durchzuführen, bietet es sich an die UpdateAction innerhalb der run() Methode der DotplotApplication aufzurufen, wobei nach einem erfolgreichen Update durch die Rückgabe von IPlatformRunnable.EXIT_RESTART ein Neustart des RCP ausgelöst wird.

Anhang I

Abbildungsverzeichnis

Abbildung 1 - Dotplotbeispiel	8
Abbildung 2 - Grundlegende Muster	9
Abbildung 3 - Störungen in Mustern	9
Abbildung 4 - Reorganisation von Blöcken und Diagonalen	10
Abbildung 5 - Kombination von Mustern	10
Abbildung 6 - Dotplot aller Werke von Shakespeare	11
Abbildung 7 - Vergleich von Übersetzungen	12
Abbildung 8 - Versionsvergleich	13
Abbildung 9 - Vergleich von 290.000 Dateinamen	14
Abbildung 10 - 2 Millionen Zeilen C-Code	15

Diagrammverzeichnis

Diagramm 1 - Struktur von Pipes & Filters	22
Diagramm 2 - Pipes & Filters in der Anwendung	23
Diagramm 3 - Konzept der Plotquellen	34
Diagramm 4 - Konzept der Typisierung	34
Diagramm 5 - Typhierarchie der neuen Architektur	35
Diagramm 6 - Konzept des Konfigurationsmanagements	36
Diagramm 7 - Übersicht Service Framework	39
Diagramm 8 - Struktur des Musters Befehl	40
Diagramm 9 - Anwendung von Befehl	41
Diagramm 10 - Struktur des Musters Strategie	42
Diagramm 11 - Verwendung von Strategie 1	43
Diagramm 12 - Verwendung von Strategie 2	44
Diagramm 13 - Struktur des Musters Fabrikmethode	45
Diagramm 14 - Fabrikmethode in der Anwendung	46
Diagramm 15 - Struktur des Musters Registry	47
Diagramm 16 - Registry in der Anwendung	48
Diagramm 17 - Struktur des Musters Plugin	55
Diagramm 18 - Plugin in der Anwendung	56
Diagramm 19 - Aufruf eines Jobs	59
Diagramm 20 - Erzeugen des DotplotContext	60
Diagramm 21 - Struktur von Interceptor	61
Diagramm 22 - Anwendung von Interceptor	63
Diagramm 23 - Aufruf der Interceptoren	64
Diagramm 24 - Paketstruktur der neuen Architektur	71

Anhang II

Literaturverzeichnis

- [DTP] Church, Kenneth Ward, Jonathan Helfman:
Dotplot: a Program for Exploring Self-Similarity in Millions of Lines Text and Code,
Journal of Computational and Graphical Statistics, Vol.2, Nr.2, 1993
- [DSP] Jonathan Helfman:
Dotplot Similarity Patterns,
<http://www.imagebeat.com/dotplot/>
- [DP] Jonathan Helfman:
Dotplot Patterns: A Literal Look at Pattern Languages,
Theory and Practice of Object Systems, Vol.2, Nr.1, 1996
- [Pla] Paul Clough:
Plagiarism in natural and programming languages: an overview of current tools and technologies,
University of Sheffield, 2000
- [EC] The Eclipse Foundation:
Eclipse.org,
<http://www.eclipse.org/>
- [RCE] Berthold Daum:
Rich-Client-Entwicklung mit Eclipse 3.1,
dpunkt, 2005
- [CVS] Derek Robert Price:
CVS - Open Source Version Control,
<http://cvs.nongnu.org/>
- [Ant] The Apache Software Foundation:
The Apache Ant Project,
<http://ant.apache.org/>
- [JU] Object Mentor, Incorporated:
JUnit, Testing Resources for Extreme Programming,
<http://www.junit.org/>
- [VgD] Tobias Gesellchen:
Visualisierung großer Datenmengen,
Diplomarbeit, Fachhochschule Gießen-Friedberg, 2005
- [SF] Matrix4.plot-Team:
SourceForge.net: DotPlot,
<http://sourceforge.net/projects/dotplot/>
- [M4DP] Matrix4.plot-Team:
Dotplot, das Eclipse-Plugin,
<http://www.dotplot.org/>

- [JFlex] Gerwin Klein:
JFlex - The Fast Scanner Generator for Java,
<http://www.jflex.de/>
- [PoSa] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal:
Patternorientierte Softwarearchitektur: Ein Pattern-System,
Addison-Wesley, 1998
- [XP] Don Wells:
Extreme Programming,
<http://www.extremeprogramming.org/>
- [Gof] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides:
Entwicklungsmuster,
Addison-Wesley, 1995
- [PEAA] Martin Fowler:
Patterns für Enterprise Application-Architekturen,
mitp, 2003
- [W3CX] World Wide Web Consortium:
Extensible Markup Language (XML),
<http://www.w3.org/XML/>
- [PoSaII] F. Buschmann, H. Rohnert, M. Stal, D. Schmidt:
Pattern-Oriented Software Architecture, Volume 2,
John Wiley & Sons, 2000
- [W3CS] World Wide Web Consortium:
XML Schema,
<http://www.w3.org/XML/Schema/>

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit selbständig verfasst und noch nicht anderweitig zu Prüfungszwecken benutzt habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und wörtliche oder sinngemäße Zitate als solche gekennzeichnet.

Lahnau, den 28. Februar 2006

Christian Gerhardt